

искусство программирования ИГР на C++

МИХАИЛ ФЛЕНОВ

Проблемы создания движка 3D-игры

Скелетная и вершинная анимация

Алгоритмы проверки столкновений

DirectInput и перемещения в 3D-мире

3D-звук



Михаил Фленов

Искусство программирования ИГР на C++

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06
ББК 32.973.26-018.1
Ф69

Фленов М. Е.

Ф69 Искусство программирования игр на C++. — СПб.: БХВ-Петербург, 2006. — 256 с.: ил.

ISBN 5-94157-832-6

Описаны современные технологии программирования 3D-игр, а также некоторые решения типичных проблем, с которыми может столкнуться программист при их разработке. В качестве практических примеров на протяжении всей книги рассматривается процесс создания простого движка игры, который использует все описываемые технологии: вершинные и пиксельные шейдеры, скелетную и вершинную анимацию, а также компоненты DirectMusic, DirectSound и DirectInput, входящие в библиотеку DirectX. Программиый код, приведенный в книге, легко адаптировать и превратить в полноценную игру. Описываемый движок очень прост, ио уииверсалеи и позволяет создавать игры любого жанра. На компакт-диске к книге содержатся листинги примеров и дополнительная информация по DirectX.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Рожко*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.07.06.
Формат 70×100$^1/_{16}$. Печать офсетная. Усл. печ. л. 20,84.
Тираж 3000 экз. Заказ № 543
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-832-6

Оглавление

Предисловие

С самого начала необходимо определиться, что именно будет рассматриваться в данной книге. Несмотря на то, что в названии присутствует слово "игра", полноценную игру мы писать не будем, потому что для этого понадобится достаточно большая книга или даже несколько книг объемом примерно в 1000 страниц. В книге на 300—400 страницах можно описать только самую простейшую игру определенного жанра и с простейшими графическими возможностями. Но не это наша цель. Наша цель рассмотреть используемые технологии, понять проблемы, с которыми может столкнуться разработчик движка игры[1] и заложить багаж знаний, который пригодится вам в будущем.

Я мог бы написать простую игру и рассказать в книге, как это было сделано, но от нее толку будет мало. Намного интереснее, если вы сами создадите что-то самостоятельно, а данная книга поможет вам сделать это. Мы рассмотрим основные проблемы, с которыми вы можете столкнуться, а превращение кода в полноценную игру будет делом техники, знаний и умений.

Если честно, то популярность игры зависит от графических возможностей не более чем на 30%. Можно привести массу примеров, когда игра становилась популярной, а использовала старинные графические движки и текстуры плохого качества. Недаром даже в те времена, когда на PC уже начал властвовать Wolfstain 3D, мы продолжали на работе раскладывать примитивный тетрис. Проходят годы и появляются более красивые игры, но даже во времена Half-Life мы с удовольствием раскладываем линии (Lines). Самое главное в игре — это сюжет и геймплей (*Геймплей* — это сущность игры — динамика,

---

[1] *Движок игры* — набор функций или классов, которые предоставляют разработчику возможность создавать игровой мир. Чаще всего в движке реализована возможность загрузки, хранения и отображения объектов игрового мира, искусственный интеллект, возможности сетевого обмена данными, поддержка музыки и звуковых эффектов и т. д.

управление, развитие, от которых зависит интерес к самой игре). Если в игру интересно играть, то она будет популярной и чаще всего это не зависит от графики, и этому есть множество подтверждений.

Сложность графики — это всего лишь вспомогательная составляющая, но зато какая приятная для игрока и интересная для программиста. Создавать красивую графику интересно, и смотреть на нее приятно.

Слишком сложная графика может наоборот отрицательно повлиять на популярность игры. Если для формирования сцены необходимо слишком много процессорных ресурсов и для ее запуска требуется компьютер, стоимостью в $3000, то игра никогда не станет популярной, потому что такими компьютерами обладают единицы и не факт, что все они заплатят за ваш труд. А если сделать игру слишком простой, чтобы она запускалась даже на Pentium с частотой 100 МГц, то на такую графику невозможно будет смотреть и с вероятностью в 99% такую игру ждет полный провал, если нет суперуникального и супервеликолепного геймплея.

Великое искусство — найти идеальное сочетание производительности и качества изображения. Тут приходится применять все возможные алгоритмы оптимизации и нередко нужно обращаться к искусству хакеров. Нет, я не призываю вас писать игру на языке ассемблера или даже в машинных кодах, чтобы она была идеально быстрой, но над оптимизацией придется хорошенечко подумать.

На протяжении всей книги мы будем писать простой движок 3D-игры, а также учиться создавать различные эффекты, которые в сочетании с хорошим геймплеем могут сделать вашу игру шедевром. Да, наработки, которые мы напишем, могут пригодиться вам в будущих проектах, и все же, их нельзя назвать идеальными.

В программировании нет предела совершенству, но к нему необходимо стремиться. Если такие слова еще никто не говорил, то запишите их на мой счет ☺, потому что это реальность программиста.

О чем эта книга

Итак, данная книга посвящена созданию графической части игр, а точнее, оптимизации, графическим эффектам и алгоритмам реализации классических задач, которые встречаются в играх. Для иллюстрации примеров мы будем использовать язык программирования C++ и графический пакет DirectX.

Какое отношение имеют хакеры к играм и графическим эффектам? Дело в том, что графические эффекты получили распространение именно благодаря хакерам. Именно они, с помощью *демо-роликов* — небольших программ, ко-

торые используют скрытые возможности компьютера для создания красивых графических эффектов. Некоторые программисты, попав в мир демо-сцен (demoscene), продолжают совершенствовать свои умения на общественных началах или просто уходят в другую область.

Если не иметь постоянного источника дохода, то такие программисты долго не держатся на сцене (не работают над созданием роликов demoscene), потому что создаваемые ролики не приносят дохода. Можно привести только единичные случае, когда создатель эффектов и роликов получил за свою работу деньги, и к таким случаям относятся:

- победа на конкурсе;
- получение заказа на создание ролика от какой-нибудь фирмы, например, демо-ролики очень часто можно встретить у таких компаний, как Nvidia, ATI и др.

Получить доход от любимого дела сложно. А ведь нужно как-то кушать самому и кормить свою семью, если она существует. Себя прокормить на пособия по безработице в Европе и штатах можно, а вот семью проблематично. Работать где-то и параллельно заниматься сценой сложно, потому что графика отнимает слишком много времени. Именно поэтому мир демо-сцен (demoscene) принадлежит молодым, и чаще всего не работающим ребятам.

Чтобы остаться в мире графики и получать за свою работу зарплату или другого вида доход, нередко программисты уходят в мир создания компьютерных игр. Именно так можно продолжать оттачивать мастерство в сфере графических эффектов, заниматься любимым делом и получать за это деньги. Это мечта любого землянина — заниматься своим любимым делом за деньги.

Мое мнение, что хакеры, которые начинали свою практику с демо-роликов, очень сильно повлияли на мир компьютерных игр и благодаря им графика становится все более реалистичной. Конечно же, не хакерами едиными живет компьютерная графика, и есть программисты, которые никак не были связаны с демо-сценой, но я думаю, что большинство из них все же что-то использовали из мира хакеров.

Более подробно о демо-сцене и о графических эффектах в роликах вы можете узнать из книг [4] или [6] ("DirectX и C++. Искусство программирования" и "DirectX и Delphi. Искусство программирования". — СПб.: БХВ-Петербург, 2006). Если вы их не читали и мало знакомы с DirectX и ее 9-й версией, то рекомендую прочитать сначала именно одну из этих книг, в зависимости от знакомого вам языка программирования — C++ или Delphi. Дело в том, что книга, которую вы держите сейчас в руках, является своего рода продолжением этих книг, и более глубоко погружает нас в мир компьютерной графики и создания эффектов.

Я буду очень часто ссылаться на книги [4] и [6], которые описывают одни и те же примеры, только для иллюстрации используется свой язык программирования — C++ или Delphi. Нет смысла покупать сразу две книги, возьмите только одну, использующую тот язык программирования, который вы знаете.

Что нужно знать

Для понимания данной книги вам потребуются знания языков C++ и DirectX. Причем глубоких знаний языка программирования C++ не нужно, потому что в работе с графикой мы практически не будем использовать MFC. В основном нам понадобится математика, циклы и условные операторы.

А вот DirectX желательно знать хорошо, в том числе и работу с шейдерами, благодаря которым можно получить наиболее реалистичные эффекты. Все, что необходимо знать из пакета DirectX, описано в книгах [4] и [6], и если вы их читали, то с пониманием этой книги вопросов не должно возникнуть. Те вопросы, которые не рассматривались в моей первой книге по графике, здесь будут освещены очень подробно. Получается, что лучше все же купить обе книги. Данная работа является как бы продолжением моей первой работы по графике — "DirectX и C++. Искусство программирования" и "DirectX и Delphi. Искусство программирования".

Что не вошло в книгу

Для создания полноценной игры вам необходимо:

- разработать сценарий;
- создать игровые уровни;
- нарисовать текстуры объектов;
- смоделировать 3D-объекты;
- написать музыку и звуковые эффекты.

Это не полный цикл создания игры, но это все и то, что упущено в списке, рассматриваться не будет. Мы будем разбирать только графическую часть игры, эффекты и оптимизацию, т. е. то, что можно позаимствовать из мира хакеров и использовать в играх.

Благодарности

Единственное, что читают жена и родители в моих книгах — благодарности. Да, родители больше ничего особо и не понимают в книгах, а жена не особо пытается. Это хорошо, потому что еще один программист в семье будет серь-

езной проблемой. Я даже не могу себе представить этот процесс. Я хочу поблагодарить их за помощь и поддержку, даже несмотря на то, что компьютер отнимает у меня слишком много времени.

Благодарю всех, кто помогает мне в работе: сотрудников издательства "БХВ-Петербург", редакторов, друзей, сотрудников журнала "Хакер" (особенно Лозовского Александра, рецензии которого находятся на задней обложке к большинству моих книг), всех друзей по сайту **www.vr-online.ru** и просто всех, кто меня знает и при этом хотя бы не ненавидит ☺.

Если у вас возникли вопросы по книге или просто пожелания, то я всегда открыт к общению. Пишите мне на адрес издательства **mail@bhv.ru** или лучше заходите на форум сайта **www.vr-online.ru**. Отвечать на почту мне удается далеко не всегда, а на форум я захожу каждый день, и по возможности стараюсь помочь всем, у кого возникли сложности или вопросы.

Если вы готовы погрузиться в мир компьютерной графики и игровых эффектов, то переворачивайте страницу, мы будем постепенно погружаться в этот мир. Надеюсь, что он вам понравится.

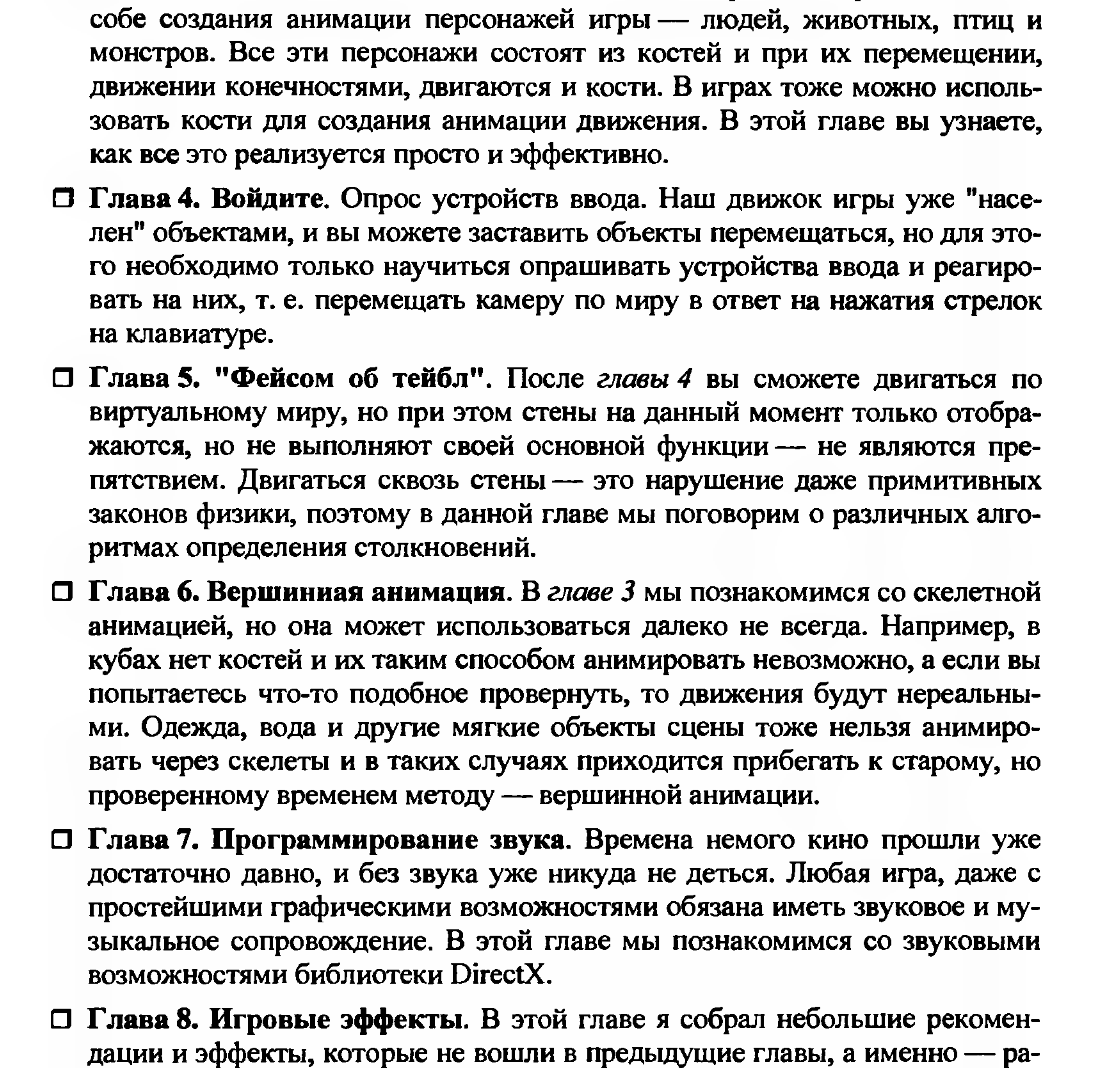
Структура книги

Я постарался сделать рассказ максимально последовательным. На протяжении следующих 8 глав вы будете последовательно писать движок игры, постепенно наращивая его возможности. Я настоятельно рекомендую, чтобы вы именно самостоятельно писали, тогда вы:

- лучше сможете понять код, лучше ориентироваться в нем или даже оптимизировать, а в некоторых главах есть что оптимизировать;
- быстро сможете адаптировать движок под собственные возможности. Движок хоть и построен специально для обучения, но может использоваться и в реальных боевых условиях.

Теперь посмотрим, что представляют собой 8 глав книги.

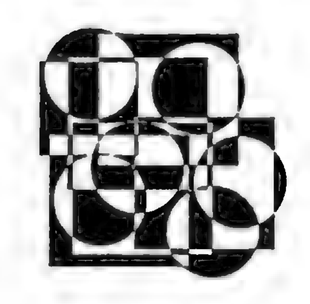
- **Глава 1. Введение в программирование игр**. В этой главе мы поговорим об основах программирования. Большую часть посвятим созданию базового приложения, которое будет использоваться в дальнейшем. Базовая часть будет уметь инициализировать Direct3D, обрабатывать события, отображать сцену и загружать сетки Mesh.
- **Глава 2. Разработка движка**. В этой главе наше базовое приложение будет обрастать возможностями. Сначала мы напишем класс, с помощью которого можно будет управлять объектами сцены, и научим наш движок отображать эти объекты с учетом освещения и даже теней. Да, о тенях нужно заботиться заранее, а не в самом конце, как делают некоторые разработчики.



- **Глава 3. Скелетная анимация.** Здесь мы поговорим о современном способе создания анимации персонажей игры — людей, животных, птиц и монстров. Все эти персонажи состоят из костей и при их перемещении, движении конечностями, двигаются и кости. В играх тоже можно использовать кости для создания анимации движения. В этой главе вы узнаете, как все это реализуется просто и эффективно.
- **Глава 4. Войдите.** Опрос устройств ввода. Наш движок игры уже "населен" объектами, и вы можете заставить объекты перемещаться, но для этого необходимо только научиться опрашивать устройства ввода и реагировать на них, т. е. перемещать камеру по миру в ответ на нажатия стрелок на клавиатуре.
- **Глава 5. "Фейсом об тейбл".** После *главы 4* вы сможете двигаться по виртуальному миру, но при этом стены на данный момент только отображаются, но не выполняют своей основной функции — не являются препятствием. Двигаться сквозь стены — это нарушение даже примитивных законов физики, поэтому в данной главе мы поговорим о различных алгоритмах определения столкновений.
- **Глава 6. Вершинная анимация.** В *главе 3* мы познакомимся со скелетной анимацией, но она может использоваться далеко не всегда. Например, в кубах нет костей и их таким способом анимировать невозможно, а если вы попытаетесь что-то подобное провернуть, то движения будут нереальными. Одежда, вода и другие мягкие объекты сцены тоже нельзя анимировать через скелеты и в таких случаях приходится прибегать к старому, но проверенному временем методу — вершинной анимации.
- **Глава 7. Программирование звука.** Времена немого кино прошли уже достаточно давно, и без звука уже никуда не деться. Любая игра, даже с простейшими графическими возможностями обязана иметь звуковое и музыкальное сопровождение. В этой главе мы познакомимся со звуковыми возможностями библиотеки DirectX.
- **Глава 8. Игровые эффекты.** В этой главе я собрал небольшие рекомендации и эффекты, которые не вошли в предыдущие главы, а именно — работа с видеороликами, которые вы можете использовать для заставок между уровнями, работа с мелкими объектами, создания различных эффектов и т. д. Эта глава подскажет вам, как сделать игру лучше и красивее.

Желаю вам приятного чтения. Надеюсь, что книга получилась хорошей, по крайней мере, мы все старались (вместе с редакторами, корректорами, дизайнерами).

ГЛАВА 1



Введение в программирование игр

Для начала нам необходимо определиться с базой, которая будет использоваться на протяжении всей книги. Напоминаю, если вы читали книгу "DirectX и C++. Искусство программирования" [4], то никаких проблем не возникнет. В некоторых случаях наши примеры будут пересекаться с данной книгой, и основа также будет схожа. Большая часть этой главы делает краткий экскурс по функциям, которые были написаны в книге, на которую я устал давать ссылку (а что поделаешь, если это продолжение), поэтому, если вы уже ознакомились с этой книгой, можно только просмотреть данную главу.

Игры, как и Demo-ролики, требуют серьезной оптимизации. Чем меньше ресурсов потребляет игра и при этом формирует качественную картинку, тем большее число пользователей сможет запустить игру. Если не обращать внимания на оптимизацию, то при максимальных возможностях и большом разрешении минимальными требованиями для игры может стать, например, Pentium 8, который может появиться лет через 5. Такую игру ожидает полный провал.

Итак, давайте потихоньку начнем погружаться в мир программирования. Для этого у вас уже должен быть установлен DirectX SDK и пути на заголовочные файлы и библиотеки должны быть уже прописаны в среде разработки. Надеюсь, вы знаете, как это делается.

1.1. Подготовка к созданию приложения

Оптимизировать игры сложнее. Если в Demo-роликах используется только графика и звук и весь код можно реализовать на чистом языке C, то в играх без объектов обойтись сложно. Объекты необходимы, но это не значит, что вы должны использовать библиотеку MFC. Из возможностей этой библиотеки и WinAPI в приложении, применяющей DirectX, необходимо только соз-

дание окна. Больше ничего из API Windows нам не поможет, и будут использоваться только интерфейсы DirectX или функции языка С. Поэтому скелет программы не должен использовать объектов. Давайте создадим скелет будущего игрового приложения.

Итак, создайте пустое приложение **File | New | Project** (Файл | Новый | Проект) и в появившемся окне выберите в дереве **Project Type** (Тип проекта) пункт **Visual C++ Projects | Win32** (Проекты Visual C++ | Win32). Чтобы наши программы были небольшими и быстрыми, мы не будем использовать MFC, поэтому выбираем пункт **Win32 Project** (Проект Win32) (рис. 1.1). В поле **Name** (Имя) укажите имя проекта, а в раскрывающемся списке **Location** (Месторасположение) — путь, по которому будет сохранен проект.

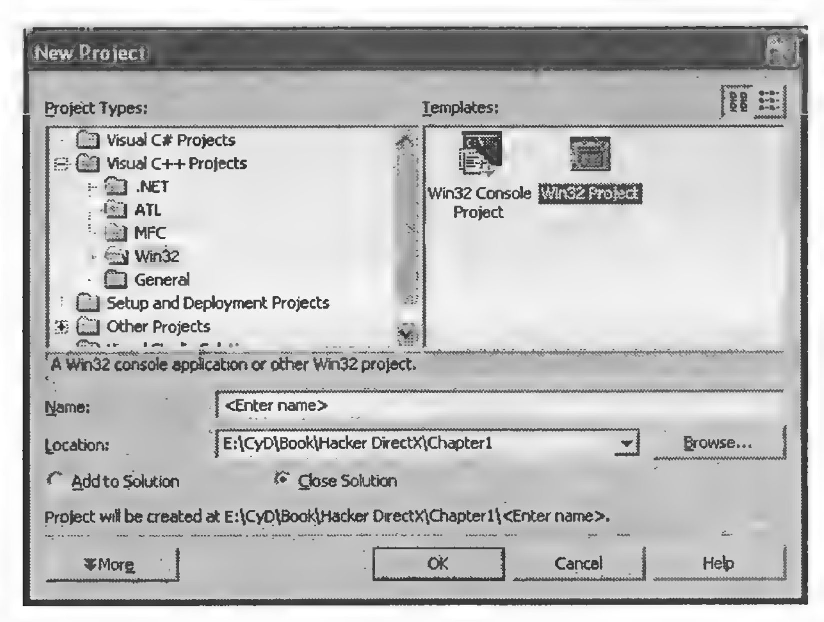


Рис. 1.1. Окно создания нового проекта

После нажатия кнопки **OK** перед нами открывается мастер создания приложения. В разделе **Application Settings** (Настройки приложения) (рис. 1.2) установите тип приложения, выбрав переключатель **Windows application** (Приложение Windows) в группе переключателей **Application type** (Тип приложения), а все прочие флажки должны остаться по умолчанию.

Теперь наше базовое приложение готово. Не забудьте в свойствах проекта подключить библиотеки Direct3D. Для этого выберите меню **Project | Properties** (Проект | Свойства). В дереве свойств выбираем раздел **Configuration Properties | Linker | Input** (Свойства конфигурации | Сборщик | Входящие). Перед вами откроется окно, как на рис. 1.3.

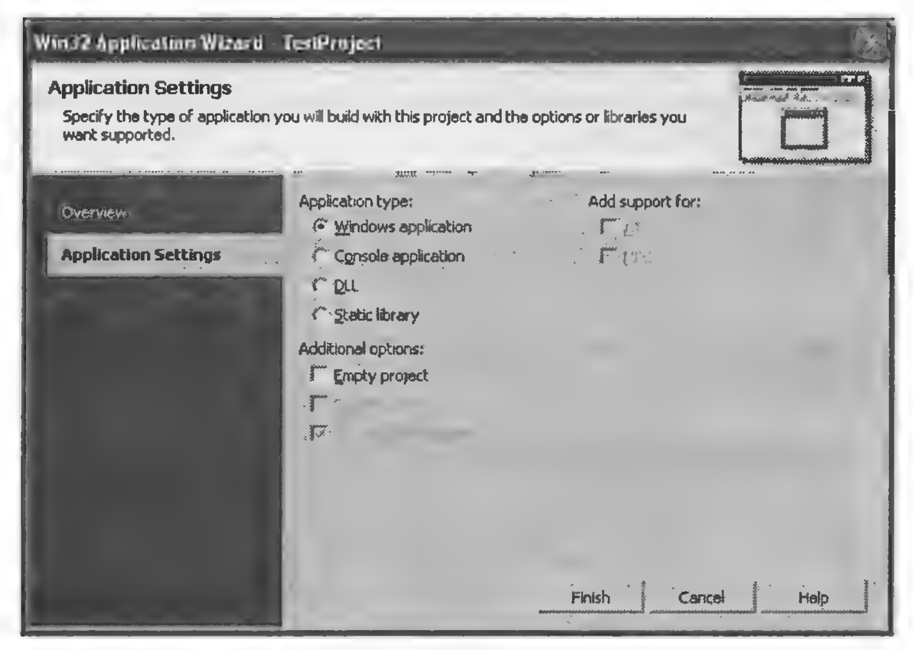


Рис. 1.2. Мастер создания приложения

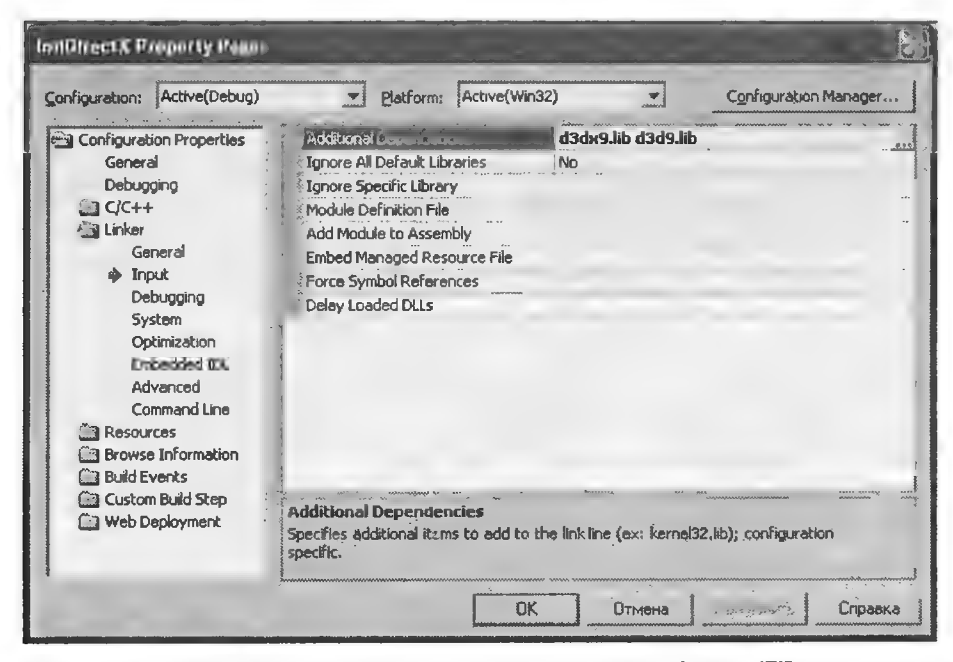


Рис. 1.3. Окно настройки свойств проекта в Visual Studio .NET

В строке **Additional Dependencies** (Дополнительные зависимости) необходимо указать библиотеки, которые нужно подключить во время сборки проекта. Выделите эту строку и щелкните по кнопке с изображением трех точек. Перед вами появится окно, в котором можно указать дополнительные библиотеки (рис. 1.4).

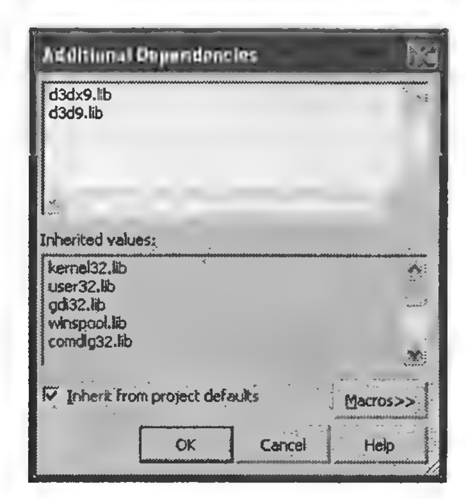


Рис. 1.4. Окно добавления библиотек

Для большинства проектов из данной книги необходимо как минимум указать библиотеки d3dx9.lib и d3d9.lib. Укажите их каждую в отдельной строке и нажмите **OK** для сохранения изменений.

Добавить библиотеки нужно для обеих конфигураций: Release и Debug. Последовательно выберите в выпадающем меню **Configuration** (Конфигурация) оба пункта и добавьте модули.

Помимо этого, в разделе **Configuration Properties | C\C++ | Precompiled Header** (Свойства конфигурации | C\C++ | Предварительно скомпилированные заголовочные файлы) для обеих конфигураций установите в параметре **Create/Use precompiled Header** (Создавать/Использовать предварительно скомпилированные заголовочные файлы) параметр **Automatically Generate (/YX)** (Автоматически генерировать).

1.2. Скелет приложения

Даже для простого приложения Win32 мастер создает слишком много лишнего. Например, меню и диалоговое окно с информацией о программе. Все это ненужно, если игра будет работать только в полноэкранном режиме. В большинстве игр меню и диалоговые окна Windows не используются, поэтому и

мы не будем этого делать. Удаляем содержимое файла, который сгенерировал мастер, и заменяем его кодом из листинга 1.1.

Листинг 1.1. Скелет игрового приложения

```
#include <windows.h>
#include "d3d9.h"
#include "d3dx9.h"
#include "..\..\common\dxfunc.h"

// Глобальные переменные
char szWindowClass[] = "Direct3DTemplateProj";
char szTitle[] = "Direct3D Demo by Michael Flenov";

// Объекты Direct3D
IDirect3D9 *pD3D = NULL;
IDirect3DDevice9 *pD3DDevice = NULL;

int iWidth=800;
int iHeight=600;

// Объявление функций
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
    LPSTR szCmdLine, int nCmdShow);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
bool Init(HWND hWnd);
void GraphEngine();

// Главная функция WinMain
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev,
   LPSTR szCmdLine, int nCmdShow)
{
 WNDCLASSEX wcex;
 MSG        msg;
 HWND       hWnd;

 CoInitialize(NULL);

 // Регистрируем класс окна
 wcex.cbSize = sizeof(wcex);
 wcex.style = CS_CLASSDC;
 wcex.lpfnWndProc = (WNDPROC)WndProc;
 wcex.cbClsExtra = 0;
```

```
 wcex.cbWndExtra = 0;
 wcex.hInstance = hInst;
 wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
 wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
 wcex.hbrBackground = NULL;
 wcex.lpszMenuName = NULL;
 wcex.lpszClassName = szWindowClass;
 wcex.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
 if(!RegisterClassEx(&wcex))
  return FALSE;

 // Создаем окно
 hWnd = CreateWindow(szWindowClass, szTitle,
      WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX, CW_USEDEFAULT,
      CW_USEDEFAULT, iWidth, iHeight, NULL, NULL, hInst, NULL);
 if(!hWnd)
  return FALSE;
 // Отображаем окно
 ShowWindow(hWnd, SW_NORMAL);
 UpdateWindow(hWnd);

 // Инициализация
 if(Init(hWnd) == TRUE)
 {
   while (true)
   {
    if (PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
    {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
     if (msg.message == WM_QUIT) break;
    }
    GraphEngine();
   }
 }

 // Очистка выделенных ресурсов
 if (pD3DDevice) {pD3DDevice= NULL; pD3DDevice=NULL;}
 if (pD3D) {pD3D= NULL; pD3D=NULL;}

 CoUninitialize();

 return 0;
}
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
    WPARAM wParam, LPARAM lParam)
{
 switch(message) {
  case WM_DESTROY:
   PostQuitMessage(0);
   break;

  default:
   return DefWindowProc(hWnd, message, wParam, lParam);
 }
 return 0;
}

bool Init(HWND hWnd)
{
 if (DX3DInitZ(&pD3D, &pD3DDevice, hWnd, iWidth, iHeight, FALSE)!=S_OK)
 {
  MessageBox(hWnd, "DirectX Initialize Error", "Error", MB_OK);
  return FALSE;
 }
 return TRUE;
}

void GraphEngine()
{
  pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
       D3DCOLOR_XRGB(0,0,0), 1.0f, 0);
  if (SUCCEEDED(pD3DDevice->BeginScene()))
  {
    pD3DDevice->EndScene();
  }

 pD3DDevice->Present(NULL, NULL, NULL, NULL);
}
```

Давайте быстренько пробежимся по этому коду. Из него убраны все функции, которые использовались для инициализации, а осталась только одна `WinMain`. Регистрация класса, создание окна и отображение окна происходит именно здесь. Чтобы сэкономить несколько десятков байт на ресурсах, я убрал загрузку строк из ресурсов (имя класса и заголовок окна), а прописал эти строки константами. Убрана загрузка акселераторов, потому что не будет меню и визуального интерфейса.

После отображения окна вызывается функция `Init`. Эта функция добавлена для удобства, и в ней будет осуществлена инициализация Direct3D. Если результат ее выполнения положителен, то инициализация прошла успешно и можно начинать основной цикл приложения. Иначе, приложение завершит работу.

Теперь посмотрим на цикл обработки сообщений:

```
while (true)
{
 // Если есть сообщение в очереди
 if (PeekMessage(&msg, NULL, NULL, NULL, PM_REMOVE))
 {
  // Обработать сообщение
  TranslateMessage(&msg);
  DispatchMessage(&msg);
  if (msg.message == WM_QUIT) break;
 }
 // Вызвать функцию движка игры
 GraphEngine();
}
```

В данном случае для обработки сообщений (в полноэкранном режиме их будет минимум) мы запускаем бесконечный цикл. Внутри цикла сначала проверяем, есть ли для нас сообщения. Если да, то обрабатываем их. После этого проверяем, если сообщение равно `WM_QUIT`, то необходимо завершить работу программы, поэтому дальнейшее выполнение цикла не имеет смысла, и мы прерываем его с помощью оператора `break`.

Если не было сообщения `WM_QUIT`, то цикл продолжит выполнение, а здесь вызывается функция `GraphEngine`. Эта функция добавлена для удобства, чтобы вынести движок игры в виде отдельной функции, и с ней удобнее было работать.

Не обойтись и без функции `WndProc`, которая необходима для обработки сообщений. Так как у нас не будет никаких пунктов меню, то обрабатываем только событие `WM_DESTROY`, чтобы сделать корректный выход из программы.

Теперь рассмотрим функцию `Init`, которая инициализирует Direct3D и задает параметры отображения по умолчанию. В шаблонном приложении она выглядит следующим образом:

```
bool Init(HWND hWnd)
{
 // Инициализация Direct3D
 if (DX3DInitZ(&pD3D, &pD3DDevice, hWnd,
        iWidth, iHeight, FALSE)!=S_OK)
```

```
  {
   MessageBox(hWnd, "DirectX Initialize Error", "Error", MB_OK);
   return FALSE;
  }

 return TRUE;
}
```

Основа функции `Init` — это вызов функции `DX3DInitZ`. Эта функция была написана в книге "DirectX и C++. Искусство программирования" [4] и для удобства вынесена в отдельный модуль dxfunc.cpp.

Примечание

Файл dxfunc.cpp можно найти на компакт-диске в каталоге Common.

1.3. Инициализация Direct3D

Функция `DX3DInitZ` универсальна и удобна. Достаточно подключить модуль к любому проекту и проинициализировать Direct3D вызовом одной функции `DX3DInitZ`. Эту функцию вы можете увидеть в листинге 1.2.

Листинг 1.2. Функция `DX3DInitZ` для инициализации Direct3D

```
HRESULT DX3DInitZ(IDirect3D9 **ppiD3D9,
  IDirect3DDevice9 **ppiD3DDevice9, HWND hWnd,
  DWORD iWidth, DWORD iHeight, BOOL bFullScreen)
{
 // Инициализация Direct3D 9-й версии
 if((*ppiD3D9 = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
  return E_FAIL;

 // Заполняем основные параметры представления
 D3DPRESENT_PARAMETERS d3dpp;
 ZeroMemory(&d3dpp, sizeof(d3dpp));
 d3dpp.BackBufferWidth = iWidth;
 d3dpp.BackBufferHeight = iHeight;
 d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
 d3dpp.EnableAutoDepthStencil = TRUE;

 // Запрос на отображение в полноэкранном режиме
 int  iRes;
 if (!bFullScreen)
  iRes=MessageBox(hWnd, "Use fullscreen mode?", "Screen",
      MB_YESNO | MB_ICONQUESTION);
```

```
else
 iRes = IDYES;

if(iRes == IDYES)
{
 // Полноэкранный режим
 // Установка параметров полноэкранного режима
 d3dpp.BackBufferFormat = D3DFMT_R5G6B5;
 d3dpp.SwapEffect = D3DSWAPEFFECT_FLIP;
 d3dpp.Windowed = FALSE;
 d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
 d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_DEFAULT;
}
else
{
 // Оконный режим
 RECT wndRect;
 RECT clientRect;

 // Корректируем размер окна, чтобы клиентская область была
 // четко указанного размера
 GetWindowRect(hWnd, &wndRect);
 GetClientRect(hWnd, &clientRect);
 iWidth  = iWidth + (wndRect.right-wndRect.left) -
          (clientRect.right-clientRect.left);
 iHeight = iHeight + (wndRect.bottom-wndRect.top) -
          (clientRect.bottom-clientRect.top);

 // Устанавливаем размеры окна
 MoveWindow(hWnd, wndRect.left, wndRect.top, iWidth, iHeight, TRUE);

 // Получить формат пиксела
 D3DDISPLAYMODE d3ddm;
 (*ppiD3D9)->GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &d3ddm);

 // Установка параметров
 d3dpp.BackBufferFormat = d3ddm.Format;
 d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
 d3dpp.Windowed = TRUE;
}

DWORD Flags= D3DCREATE_MIXED_VERTEXPROCESSING;

// Создать 3D-устройство
HRESULT hRes;
```

```
 if(FAILED(hRes = (*ppiD3D9)->CreateDevice(D3DADAPTER_DEFAULT,
    D3DDEVTYPE_HAL, hWnd, Flags, &d3dpp, ppiD3DDevice9)))
  return hRes;

 // Установить перспективу
 float Aspect = (float)d3dpp.BackBufferWidth /
       (float)d3dpp.BackBufferHeight;
 D3DXMATRIX matProjection;
 D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI/4.0f, Aspect,
        2.0f, 1000.0f);
 (*ppiD3DDevice9)->SetTransform(D3DTS_PROJECTION, &matProjection);

 // По умолчанию освещение будет отключено
 (*ppiD3DDevice9)->SetRenderState(D3DRS_LIGHTING, FALSE);

 return S_OK;
}
```

Бегло просмотрим и эту функцию. В качестве параметров она получает:

- переменную для хранения интерфейса Direct3D;
- переменную для хранения интерфейса устройства Direct3D;
- идентификатор окна;
- ширину окна;
- высоту окна.

Кроме того, надо учитывать, что если последний параметр равен `true`, то необходимо использовать полноэкранный режим, иначе оконный.

В самом начале функции инициализируется интерфейс Direct3D. Затем заполняются параметры представления (структура `D3DPRESENT_PARAMETERS`), которая необходима для инициализации устройства Direct3D. Значения некоторых параметров этой структуры будут отличаться в зависимости от оконного или полноэкранного режима. Одинаковыми будут только ширина и высота заднего буфера. Заполнив эти параметры, отображаем на экране запрос — нужно ли использовать полноэкранный режим.

Далее идет уже разделение кода на две части. Сначала заполняются параметры для полноэкранного режима, а затем для оконного.

Когда структура `D3DPRESENT_PARAMETERS` готова, вызываем метод `CreateDevice` для создания устройства. Тут у нас указываются следующие параметры:

- адаптер будет по умолчанию;
- использовать аппаратные возможности видеокарты;

- в качестве окна будет использоваться идентификатор, который передан в качестве параметра;
- дополнительные флаги. Здесь передаем переменную Flag, которая содержит значение D3DCREATE_MIXED_VERTEXPROCESSING, т. е. используется смешанный режим обработки вершин;
- заполненная структура D3DPRESENT_PARAMETERS;
- переменная, в которую будет записан результат, т. е. указатель на устройство Direct3D.

Далее идет настройка матрицы проекции. В качестве соотношения сторон берем соотношение ширины и высоты экрана. Последняя строка кода отключает освещение. По умолчанию источники света отключены. Необходимо включать освещение только тогда, когда свет действительно нужен. Хотя в играх для улучшения реалистичности свет нужен всегда, я его отключил. Почему? Да потому что максимальную реалистичность может дать использование шейдеров, что мы и будем делать в примерах данной книги. Да, алгоритм освещения я буду использовать максимально простой, но вы легко сможете его заменить на что-то более качественное и подходящее для данной атмосферы.

1.4. Функция формирования сцены

Теперь посмотрим на функцию GraphEngine, где будет формироваться графика. Функция выглядит следующим образом:

```
void GraphEngine()
{
 // Здесь необходимо произвести предварительные расчеты
 ...

 // Очистить буфер
 pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
    D3DCOLOR_XRGB(0,0,0), 1.0f, 0);
 if (SUCCEEDED(pD3DDevice->BeginScene()))
 {
  // Здесь необходимо формировать сцену
  ...
  ...

  // Завершаем формирование сцены
  pD3DDevice->EndScene();
 }

 pD3DDevice->Present(NULL, NULL, NULL, NULL);
}
```

В самом начале функции необходимо произвести предварительные расчеты. Затем вызывается метод `Clear` интерфейса устройства Direct3D для очистки заднего буфера. Затем вызывается метод `BeginScene`, и если все прошло удачно, то можем формировать сцену. Формирование заканчивается вызовом метода `EndScene`. Теперь необходимо отобразить сцену на экране. Для этого вызываем метод `Present`.

Примечание

Исходный код проекта можно найти на компакт-диске в каталоге Chapter1/TemplateProj.

1.5. Функция загрузки сетки

В книге "DirectX и C++. Искусство программирования" [4] была написана еще одна очень удобная функция, которую будем использовать и сейчас — `LoadMesh`. Эта функция удобна для загрузки сеток Mesh из X-файлов. Эту функцию вы можете также найти в файле dxfunc.cpp и в листинге 1.3.

Листинг 1.3. Функция загрузки сеток

```
DWORD LoadMesh (char *filename, IDirect3DDevice9 *ppiD3DDevice9,
   ID3DXMesh **ppMesh, LPDIRECT3DTEXTURE9 **pMeshTextures,
   char *texturefilename, D3DMATERIAL9 **pMeshMaterials)
{
 LPD3DXBUFFER pD3DXMtrlBuffer;
 DWORD dwNumMaterials;

 // Загрузка сетки из файла
 D3DXLoadMeshFromX(filename, D3DXMESH_SYSTEMMEM,  ppiD3DDevice9,
    NULL, &pD3DXMtrlBuffer, NULL, &dwNumMaterials, ppMesh);

 // Получаем указатель на материалы
 D3DXMATERIAL* d3dxMaterials =
    (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();

 // Переменные для хранения массивов текстур и материалов
 (*pMeshTextures) = new LPDIRECT3DTEXTURE9[dwNumMaterials];
 (*pMeshMaterials) = new D3DMATERIAL9[dwNumMaterials];

 // Запускаем цикл заполнения массивов
 for( DWORD i=0; i<dwNumMaterials; i++ )
```

```
{
  // Копируем материал
  (*pMeshMaterials)[i] = d3dxMaterials[i].MatD3D;

  // Создаем текстуру
  if( FAILED(D3DXCreateTextureFromFile(ppiD3DDevice9,
     d3dxMaterials[i].pTextureFilename, &(*pMeshTextures)[i])))
   if( FAILED(D3DXCreateTextureFromFile(ppiD3DDevice9,
     texturefilename, &(*pMeshTextures)[i])))
    (*pMeshTextures)[i] = NULL;
 }

 // Возвращаем количество материалов в загруженной сетке
 return dwNumMaterials;
}
```

Коротко "пробежимся" по этой функции. В самом начале загружаем сетку из X-файла с помощью функции `D3DXLoadMeshFromX`. Эта функция выглядит следующим образом:

```
HRESULT D3DXLoadMeshFromX(
    LPCTSTR pFilename,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER* ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD* pNumMaterials,
    LPD3DXMESH* ppMesh
);
```

Посмотрим параметры функции:

- `pFilename` — имя загружаемого файла;
- `Options` — опции, которые будут использоваться при загрузке. Опций достаточно много, но мы пока будем использовать только `D3DXMESH_SYSTEMMEM`, что означает, что загрузка должна происходить в системную память;
- `pDevice` — указатель на интерфейс `IDirect3DDevice9`;
- `ppAdjacency` — указатель на буфер для смежных граней;
- `ppMaterials` — указатель на буфер для материалов;
- `ppEffectInstances` — указатель на буфер для экземпляров эффектов;

- pNumMaterials — через этот параметр будет получено количество загруженных материалов;
- ppMesh — непосредственно указатель на объект сетки.

Функция D3DXLoadMeshFromX загружает только сетку, из которой состоит объект, материалы и имена текстур. Сами текстуры будут храниться в отдельном бинарном файле. Да и сами материалы находятся в не очень удобном виде и их лучше перебросить в отдельный массив типа D3DXMATERIAL.

Итак, после загрузки сетки создаем два массива: pMeshTextures и pMeshMaterials. В первый будем сохранять текстуры, а во второй материалы. Далее запускается цикл, в котором заполняются массивы. Для загрузки текстуры из файла используем функцию D3DXCreateTextureFromFile, которая выглядит следующим образом:

```
HRESULT D3DXCreateTextureFromFile(
    LPDIRECT3DDEVICE9 pDevice,
    LPCTSTR pSrcFile,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```

Здесь у нас имеется три параметра:

- pDevice — указатель на интерфейс IDirect3DDevice9, с которым должна быть связана текстура;
- pSrcFile — имя файла;
- ppTexture — указатель на интерфейс типа IDirect3DTexture9, куда будет загружена текстура.

Если во время загрузки произошла ошибка, то пытаемся загрузить файл, который был указан в качестве параметра функции. Если и тут произошла ошибка, то текстуры не будет.

1.6. Пример загрузки сетки

Теперь посмотрим, как можно использовать функцию LoadMesh. Загрузите шаблонный проект, на основе которого мы рассмотрим загрузку сетки. Для начала объявим необходимые глобальные переменные, а понадобится нам следующее:

```
DWORD dwNumMaterials;                 // Для хранения количества материалов
ID3DXMesh *pMesh;                     // Для хранения сетки
LPDIRECT3DTEXTURE9 *pMeshTextures;    // Текстуры
D3DMATERIAL9 *pMeshMaterials;         // Материалы
```

Теперь переходим в функцию `Init`, где происходит инициализация проекта, и после инициализации Direct3D добавляем следующий код:

```
dwNumMaterials = LoadMesh("tiny.x", pD3DDevice,
          &pMesh, &pMeshTextures,  "texture.bmp", &pMeshMaterials);

// Для примера создадим источник света Direct3D
D3DLIGHT9 light;
ZeroMemory(&light, sizeof(D3DLIGHT9));
light.Type = D3DLIGHT_DIRECTIONAL;
light.Direction = D3DXVECTOR3(0.5f, 0.0f, 0.5f);
light.Diffuse.r = light.Diffuse.g = light.Diffuse.b = light.Diffuse.a =
1.0f;
pD3DDevice->SetLight(0, &light);
pD3DDevice->LightEnable(0, TRUE);
```

В этом примере загружается файл tiny.x, который идет в поставке DirectX SDK и специально предназначен для того, чтобы тестировать свои приложения. После загрузки сетки инициализируем освещение, чтобы загружаемый объект не был абсолютно черным, и вы увидели его во всей красе. В данном случае используется освещение Direct3D, дабы не усложнять сейчас пример, но в будущем мы перейдем на использование шейдеров.

Теперь посмотрим, как можно отобразить загруженную сетку. Конечно же, это необходимо делать в функции `GraphEngine`, код которой для данного примера вы можете увидеть в листинге 1.4.

Листинг 1.4. Пример отображения сетки

```
void GraphEngine()
{
 // Очистить буферы
 pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(255,255,255), 1.0f, 0);

 if (SUCCEEDED(pD3DDevice->BeginScene()))
 {
  // Увеличиваем угол поворота сцены
  ViewAngle+=0.01f;
  float b = ViewAngle;

  // Рассчитываем мировую матрицу для определения положения объекта
  D3DMATRIX World = {
    cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
    -sin(b), cos(b), 0, 0,
```

```
    -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
    0, 0,  700, 1,
  };
  pD3DDevice->SetTransform(D3DTS_WORLD, &World);

  // Включаем освещение
  pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);

  // Запускаем цикл отображения
  for (DWORD i=0; i<dwNumMaterials; i++)
  {
    // Устанавливаем материал
    pD3DDevice->SetMaterial(&pMeshMaterials[i]);

    // Устанавливаем текстуру
    if (pMeshTextures[i])
     pD3DDevice->SetTexture(0, pMeshTextures[i]);

    // Отображаем очередную часть сетки
    pMesh->DrawSubset(i);
  }

  pD3DDevice->EndScene();
 }

 // Отображаем сцену
 pD3DDevice->Present(NULL, NULL, NULL, NULL);
}
```

Результат работы программы можно увидеть на рис. 1.5. Программирую графику уже давно, но до сих пор поражаюсь, как просто создать такую красоту. Конечно, красота зависит от содержимого X-файла, но и DirectX вносит свою красоту в эту сцену.

Теперь посмотрим на код отображения. После начала формирования сцены увеличиваем счетчик `ViewAngle`, который определяет угол поворота фигуры. Да, она будет вращаться, чтобы ее можно было разглядеть с разных сторон. После этого включаем освещение.

Далее идет самое интересное — цикл отображения сетки. Сетка может состоять из множества частей и каждая из них может иметь свою текстуру и материал. Количество шагов цикла — это количество загруженных материалов. В идеале, каждая фигура должна быть из одного материала/текстуры, чтобы цикл не выполнялся слишком долго и не создавал излишних накладных расходов.



Рис. 1.5. Результат работы примера

На каждом этапе цикла устанавливаем материал, и если существует текстура, то устанавливаем и ее. После этого отображаем текущую сетку с помощью вызова метода `DrawSubset` интерфейса `ID3DXMesh`.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter1/Mesh.

1.7. Моделирование

В играх сложно обойтись без объектов сеток. Создавать объекты игры вручную из треугольников достаточно утомительная задача. Намного проще смоделировать объекты с помощью 3D-редактора и потом загрузить их в своей программе. Хороший 3D-редактор с помощью визуальных средств позволяет легко и быстро создавать достаточно сложные модели. Я предпочитаю использовать 3D Studio Max (**http://www.discreet.com**). Да, это дорогой продукт, но универсальный и позволяет создать одну и ту же фигуру разными способами.

Создав сетку персонажа или другого объекта игры, скелет и анимацию, все это необходимо сохранить для последующей загрузки из собственной программы. Вот тут возникает проблема выбора — как сохранить? Формат MAX позволяет хранить всю необходимую информацию, но он слишком сложный для использования в собственных программах. Можно использовать старый формат 3DS, но тут возникает проблема с сохранением информации о костях и самой скелетной анимации, которую мы рассмотрим в *главе 3*. Эти проблемы можно обойти, но проблематично. Придется писать собственные анализаторы и загрузки данных, а потом еще создавать классы или функции для поддержки всего загруженного в программе.

Зачем выдумывать себе проблемы, когда можно использовать Х-формат файла, который разработан Microsoft и является открытым, универсальным и легко расширяемым. В предыдущем разделе мы увидели, как легко можно загрузить и отобразить сетку на экране. В дальнейшем вы убедитесь, что Х-формат файла удобен и для создания анимации, такой как скелетная.

Итак, мы можем преобразовать созданную модель в старый формат 3DS и потом конвертировать ее в Х-файл. Для этого в Интернете можно найти множество различных утилит, от простых и бесплатных до сложных и дорогих. Для простых сцен я использую утилиту conv3ds.exe (она идет в поставке с DX SDK). Для конвертирования достаточно выполнить команду:

```
conv3ds.exe filename.3ds
```

И в результате мы получим в том же каталоге файл filename.x. Но иногда эта простая утилита конвертирует не корректно, поэтому приходится прибегать к более сложной утилите 3D Exploration (**http://www.xdsoft.com/explorer/**), которая поддерживает большинство распространенных графических форматов. Главное окно программы можно увидеть на рис. 1.6.

Сразу же бросается в глаза, что панель управления у программы скудная. Да, возможностей не так уж и много. Основанные функции программы это:

- просмотр 3D-файлов. Программа поддерживает все распространенные форматы, в том числе и игровые, такие как файлы моделей для игр Quake и Quake 2. Поддержка игровых форматов очень полезна для нас, как для разработчиков 3D-игр и аниматоров;
- конвертирование из одного формата в другой. Для решения этой задачи у программы есть множество настроек. Количество форматов файлов, в которые вы можете конвертировать 3D-сцену, намного меньше, но основные форматы есть.

Функции по редактированию сцены очень скудные, да и не это главное.

Если я не ошибаюсь, то корни у этой программы русские. По крайней мере, мне так сказал мой канадский корректор, который корректировал мой плохой

английский в моих программах и на сайте. Этот корректор работал и с разработчиком программы 3D Exploration. Но это так, отступление от жизни. Если корни действительно русские, то стоит гордиться людьми, которые могут создавать такие хорошие проекты и чего-то добиваться своим трудом.

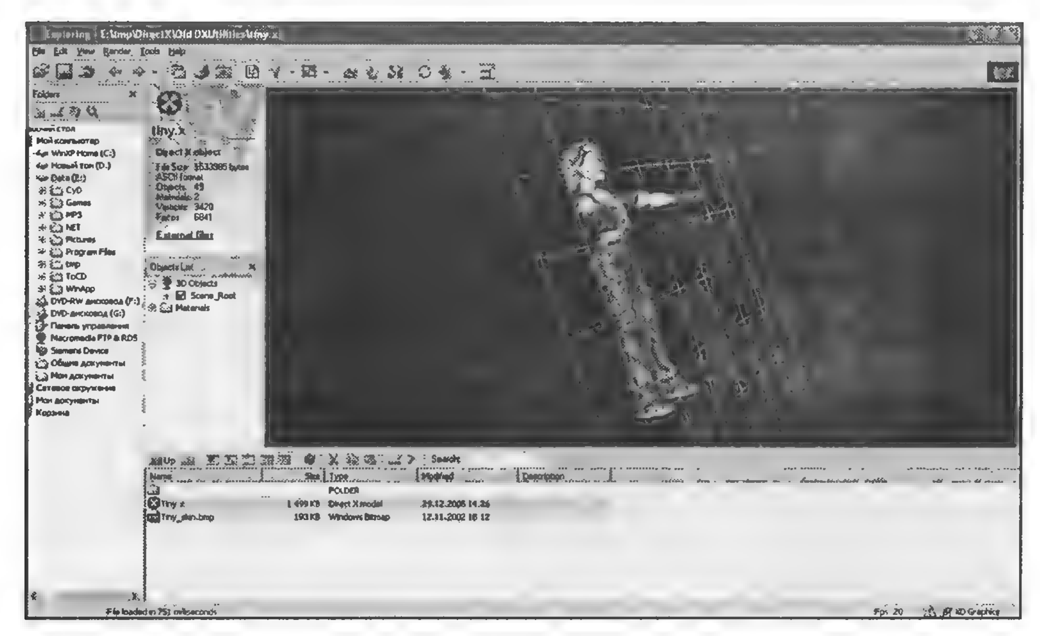


Рис. 1.6. Главное окно прогрвммы 3D Exploration

Я пытался связаться с разработчиком и получить разрешение выложить его программу на диск к книге, но ответа так и не получил.

Расширять возможности файла нам не понадобится, потому что все необходимое для хранения сетки и костей в нем уже есть. Сама 3ds Max не может сохранять или экспортировать данные в X-формат, но можно установить специальный подключаемый модуль (plug-in), разработанный самой Microsoft. Этот plug-in можно найти в составе DX SDK, причем в исходных кодах. Нам остается только скомпилировать его и установить. На рис. 1.7 можно увидеть, как выглядит этот подключаемый модуль в программе 3DS.

Хватит рекламировать 3D Studio Max, к тому же Autodesk (владелец торговой марки) не платит мне за это. Это только в кино, когда показывают какой-то продукт, то это скорей всего является скрытой рекламой. А мы этим заниматься не будем, поэтому выбирайте то, что удобно и по карману именно вам. Я только высказал свои предпочтения и показал, как можно конвертировать проекты 3D Studio Max в X-формат.

Итак, с подготовительными "телодвижениями" покончено, давайте переходить к более интересным вещам, а именно — к работе с графикой.

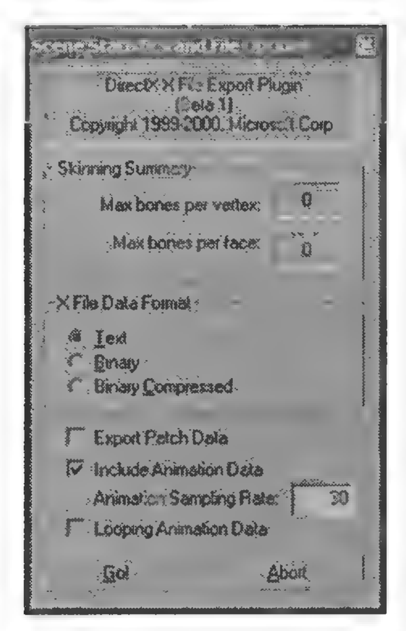


Рис. 1.7. Панель подключаемого модуля для выгрузки объекта в Х-формат

1.8. Шейдеры

В 3D-мире есть очень хороший способ создания сложных сцен, которые поразят ваше воображение. Для этого необходимо всего лишь воспользоваться *шейдерами*. Что это такое? Это небольшая программа, которая позволяет управлять регистрами видеокарты, благодаря чему можно создать невероятные по красоте эффекты. Для написания таких программ используется специализированный *язык HLSL* (High Level Shader Language, высокоуровневый язык шейдеров) или ассемблер.

В данной книге для создания таких эффектов, как освещение и тени, мы будем использовать шейдеры. Эта глава предназначена для тех, кто не знаком с этим языком, и здесь мы кратко "пробежимся" по основам. Существует два типа шейдеров:

- вершинный;
- пиксельный.

Давайте рассмотрим, что они представляют и как используются.

Вершинный шейдер, как следует из названия, предназначен для работы с вершинами. Программа шейдера выполняется для каждой вершины во время ее обработки графической картой. Здесь можно изменять цвета вершин, ко-

ординаты текстуры и т. д. Все это мы можем делать в программе с помощью интерфейсов Direct3D, а можем возложить и на вершинный шейдер.

Возлагать расстановку пикселов на шейдер более эффективно. Допустим, что в вашей сцене есть 1000 вершин, с помощью которых создаются капли дождя. Чтобы отобразить это количество вершин, необходимо написать и выполнить примерно следующий код:

```
for (int i=0; i<1000; i++)
 {
  Установить мировую матрицу, чтобы определить
  позицию капли в виртуальном мире

  Отобразить вершину
 };
```

В цикле всего два действия, но зато какие! Тысячу раз устанавливать мировую матрицу для определения положения вершины — это достаточно накладные расходы. Намного эффективнее воспользоваться шейдером. В этом случае вы должны создать массив из координат точек и отправить их на отображение. Шейдер позиционирует точки намного быстрее, а наша программа избавится от цикла.

Пиксельный шейдер используется при формировании фрагментов (пикселов) сцены. С помощью пиксельного шейдера вы можете попиксельно воздействовать на сцену, создавая эффекты "сумасшедшей сложности".

Если раньше в компьютерных играх для создания теней и освещения использовались ручные расчеты, а сцена формировалась, как и десятки лет назад, с помощью рисования трехмерного мира на двумерной плоскости, то теперь подобные эффекты легко реализовать с помощью пиксельных шейдеров.

Изначально шейдеры приходилось писать на языке ассемблера, потому что их код выполняется видеокартой, но это достаточно сложно. Если вы знакомы с этим языком, то, наверно, можете представить себе, как будет выглядеть функция преобразования матриц. А ведь без матриц в трехмерной графике никуда.

В настоящее время для работы с шейдерами используется язык HLSL. Давайте немного познакомимся с языком HLSL, потому что он понадобится уже в ближайшее время, а точнее, в следующей главе.

Все программы, которые не используют шейдеры, могут великолепно работать даже на старых видеоускорителях, разработанных в конце 90-х годов. Эти ускорители до сих пор могут стоять в пользовательских компьютерах (например, Riva TNT), и вы должны учитывать, что в них нет поддержки вершинного или пиксельного шейдера. Даже в современных видеокартах шей-

2 Зак. 543

деры могут не работать. Например, на моем ноутбуке установлен видеочип Extrime Graphics от Intel. Чип хороший, но шейдеры не воспринимает, поэтому некоторые примеры на подобных компьютерах работать просто не будут.

Тем не менее познакомиться с этим языком никому не помешает. Тем более что к моменту выхода книги современные видеоускорители от ATI и NVIDIA получат еще большее распространение, а к тому моменту, как она попадет к вам в руки, вы прочитаете ее и напишете свою игру или демо-ролик, видеокарты с поддержкой шейдеров станут еще более распространенными и будут установлены в большинстве компьютеров.

Если немного перефразировать слова самого богатого человека планеты, то получится, что современный бизнес движется со скоростью мысли. Пока мы раздумываем, что будет завтра, это завтра уже наступает и охватывает всю планету. Не удивлюсь, если уже к выходу книги технология HLSL устареет. Хотя нет, она достаточно перспективна, и скорей всего просто появится новая, более совершенная версия шейдеров. Даже уже сейчас существует несколько версий.

Как и у любого другого языка, в HLSL есть свои типы данных. Основным является вещественное число типа `float`, размером в 32 бита, и этот тип должен быть во всех видеокартах. Графика — достаточно точная "наука", поэтому и расчеты должны быть точными, следовательно основная единица измерения с плавающей точкой. Остальные типы могут и отсутствовать, но будут эмулироваться через тип `float`. Следующий пример показывает, как можно объявить вещественную переменную `Variable`:

```
float Variable;
```

Пока все идентично описанию переменных в C++. Переменная может быть статичной (`static`), внешней (`extern`), форменной (`unoform`) или константой (`const`):

```
extern float a;
static float b;
const float c;
unoform float d;
```

Ключевое слово `extern` говорит о том, что переменная `a` может изменяться или читаться приложением с помощью методов `GetVertexShaderConstantx` и `SetVertexShaderConstantx`.

Переменную `b` изменить или прочитать из приложения нельзя, потому что она является статичной. Зато программа шейдера может использовать ее. Если необходимо, чтобы приложение не могло изменять и повлиять на какую-то переменную и соответственно нарушить тем самым работу программы шейдера.

Переменная `c` является константой и ее изменить не может никто. Последняя переменная `d` объявлена как `unoform`. Такая переменная может быть изменена только между вызовами функций рисования.

В языке HLSL есть еще следующие простые типы данных:

- `double` — вещественное число в 64 бита;
- `half` — вещественное число в 16 бит;
- `int` — целое число;
- `bool` — булево значение.

Как видите, имена типов очень похожи на те, что есть у языка C++. Это простые типы данных, но ни для одного языка программирования этого не будет достаточно. Например, для нас, как для программистов графических приложений, очень важны матрицы и векторы. Для объявления каждого из этих типов есть несколько вариантов. Например, самый распространенный способ объявить матрицу:

```
типMxN переменные
```

Сначала пишется тип данных, а потом размерность. Значения `M` и `N` определяют размеры матрицы. Чаще всего мы работаем с матрицами 4×4, где каждый элемент является вещественным значением. Такая матрица может быть объявлена следующим образом:

```
float4x4 Matrix;
```

В этом примере объявлена матрица с именем `Matrix`, размером 4×4, и позволяющая хранить вещественные числа.

Теперь посмотрим на вектор. Что такое вектор? На самом деле, это всего лишь одномерный массив значений определенного типа. Чаще всего его объявляют как:

```
float4 Vector;
```

Здесь объявлена векторная переменная `Vector`, состоящая из 4-х значений. Тоже самое на языке HLSL можно написать еще и следующим образом:

```
float Vector[4];
```

Инициализация переменных происходит так же, как и на C++, поэтому тут лишние пояснения не нужны. Увидим все на примере. Математические операторы тоже ничем не отличаются, и вы можете использовать операторы сложения, вычитания, деления и т. д. точно так же, как и в C++.

Теперь переходим к структурам. Они тоже похожи на C++. В общем виде это выглядит так:

```
struct имя {
 Описание полей структуры
}
```

Следующий пример показывает структуру VS_OUTPUT:

```
struct VS_OUTPUT {
 float4 pos :POSITION;
 float4 col :COLOR;
};
```

При работе с шейдерами нужно быть внимательным, дело в том, что существует несколько версий вершинных шейдеров и каждая из них отличается возможностями и количеством поддерживаемых команд. При этом не каждая видеокарта может поддерживать запрашиваемую вами версию. До версии 1.4 шейдеры поддерживают только до 12-ти команд на всю программу. Для начала нам будет достаточно, но если нужно больше команд, то придется выбирать версию шейдера поновее, например версию 1.4 (28 команд и 8 констант), или переходить на ветку 2.0.

На этом пока закончим рассмотрение теории шейдеров и перейдем к практической части. О языке HLSL можно писать отдельную книгу, и полное описание функций языка отнимет не менее 200 страниц, а на описание примеров для закрепления теории уйдет в два раза больше.

1.8.1. Простейший пример шейдеров

Давайте сначала напишем код шейдера, а потом уже посмотрим, как его использовать в программе. Итак, код этого шейдера показан в листинге 1.5.

Листинг 1.5. Шейдер, изменяющий положение и цвет вершин

```
float4x4 mat : WORLDVIEWPROJECTION;
float TimeFactor = 1.0f;

// Структура, описывающая выходные данные
struct VS_OUTPUT {
    float4 pos  : POSITION;
    float4 col : COLOR0;
};

// Функция шейдера
VS_OUTPUT ShaderFunc(float4 Pos : POSITION){
    VS_OUTPUT Out = (VS_OUTPUT)0;

    // Установка местоположения
    Out.pos = mul(Pos, mat);
```

```
    // Установка цвета
    Out.col.r = TimeFactor;
    Out.col.g = Out.col.r/10;
    Out.col.b = 1- Out.col.g;

    return Out;
}

// Функция отображения
technique Transform {
    pass P0 {
        VertexShader = compile vs_1_1 ShaderFunc();
    }
}
```

Весь этот код необходимо сохранить в файле simple.vsh. Впоследствии мы загрузим его из программы на C++ и используем для вывода куба.

Теперь посмотрим, что у нас тут происходит. В самом начале объявляются две глобальные переменные:

```
float4x4 mat : WORLDVIEWPROJECTION;
float TimeFactor = 1.0f;
```

Первая переменная имеет тип матрицы размером 4×4 и состоит из вещественных чисел. Обратите внимание, что после объявления переменной стоит двоеточие и ключевое слово `WORLDVIEWPROJECTION`. Это даже выражение, которое состоит из трех слов `WORLD`, `VIEW` и `PROJECTION`, т. е. из трех матриц: мировой, просмотра и проекции.

Следующая переменная `TimeFactor`, которая имеет тип `float` и ей сразу же присваивается значение 1.0.

Вторая переменная имеет значение по умолчанию, поэтому программа не обязана изменять его. Матрица положения не имеет значений, поэтому тут необходимо указать значение, прежде чем вызывать код шейдера, иначе могут возникнуть проблемы.

Следующим этапом объявляется структура `VS_OUTPUT`:

```
struct VS_OUTPUT {
    float4 pos  : POSITION;
    float4 col : COLOR0;
};
```

Эта структура имеет определенное значение, я привык давать ей имя `VS_OUTPUT`, но это не есть обязательно. Такая структура используется для свя-

зи входящих и выходящих вершин, т. е., устанавливая значения полей структуры, мы влияем на результирующее состояние вершины.

Структура состоит из двух полей типа float4, т. е. из двух векторов, по 4 значения в каждом. Первая переменная с именем pos — это вектор позиции вершины. Об этом говорит ключевое слово (модификатор) POSITION после объявления переменной. Вторая переменная col — это цвет вершины, об этом говорит ключевое слово (модификатор) COLOR0. Получается, что через переменные pos и col мы сможем устанавливать цвет и позицию вершины.

Имена переменных могут быть любыми, а вот тип и модификатор обязательно должны быть такими. Если попытаться определить позицию через переменную float, а не вектор float4, то произойдет ошибка. Одним значением определить позицию вершины в мировом пространстве невозможно. Для 3D-мира нужно как минимум три значения.

Теперь объявляем функцию с именем ShaderFunc:

```
VS_OUTPUT ShaderFunc(float4 Pos : POSITION)
```

Имя может быть любым, а вот возвращаемое значение и передаваемые параметры должны быть именно такими. Что возвращает функция? А возвращает она структуру типа VS_OUTPUT, а на входе она получает векторную переменную Pos типа float4. Функция получает в качестве параметра позицию вершины, а в результате должна вернуть структуру VS_OUTPUT, которую мы только что рассматривали.

Получив текущую позицию вершины, функция может изменить ее и, назначив цвет, вернуть результат через структуру VS_OUTPUT. Установленные нами в функции значения будут использоваться видеокартой при формировании сцены. Данная функция должна вызываться для каждой вершины, из которых будет состоять отображаемый через шейдер объект.

Теперь посмотрим на код функции. В самом начале объявляется переменная Out, которая имеет тип VS_OUTPUT:

```
VS_OUTPUT Out = (VS_OUTPUT)0;
```

Через эту переменную мы сформируем значение, которое нужно вернуть. Давайте посмотрим, как формируются значения возвращаемой структуры. Сначала мы устанавливаем позицию с помощью функции mul:

```
Out.pos = mul(Pos, mat);
```

Функция mul перемножает два значения, указанных в качестве параметра и возвращает результат. В качестве параметров могут быть скаляр (число), вектор или матрица. В данном случае мы перемножаем текущую позицию вершины с матрицей положения, получая, таким образом, положение вершины

в 3D-мире с учетом преобразований, которые накладываются матрицами WORLD, VIEW и PROJECTION. Именно эти матрицы преобразования передаются нам через переменную mat.

При работе с функцией mul нужно быть очень осторожным. Дело в том, что несмотря на операцию перемножения, от перемены мест параметров результат меняется. Это матрицы и здесь лучше ничего не менять местами, иначе результат будет неожиданным. Например, результат вызова функции mul(Pos, mat) даст один результат, а mul(mat, Pos) — другой.

Теперь задаем цвет вершин:

```
Out.col.r = TimeFactor;
Out.col.g = Out.col.r/10;
Out.col.b = 1- Out.col.g;
```

За цвет у нас отвечает поле col структуры Out. Это поле также является структурой, которое состоит из трех значений: r, g и b (красный, зеленый и голубой цвет вершины). Значение красного цвета будет равно переменной TimeFactor, зеленая составляющая будет равна красной, деленной на 10, а голубая равна единице минус зеленая составляющая.

Можно было бы явно задать цвет вершины, но мы рассчитываем его на основе значения переменной TimeFactor, чтобы во время анимации происходило не только вращение куба, но и плавное изменение цвета.

Если вы хотите явно задать составляющие цвета, то можно указать их значения напрямую. Например, следующий код задает красный цвет:

```
Out.col.r = 1;
Out.col.g = 0;
Out.col.b = 0;
```

Каждая составляющая цвета изменяется от 0 до 1. Единица соответствует полной яркости. В нашем случае ярким является красный цвет, а остальные просто отсутствуют.

Мы привыкли работать с цветом, когда он изменяется от 0 до 255 (или FF в шестнадцатеричном исчислении). Преобразовать в диапазон от 0 до 1 не так уж сложно:

```
Out.col.r = 1/255*r;
Out.col.g = 1/255*g;
Out.col.b = 1/255*b;
```

В данном случае r, g и b — это значения каждой из составляющих в диапазоне от 0 до 255. Чтобы их установить, мы сначала делим 1 на 255, а потом умножаем на значение переменной r, g или b соответственно.

Самое последнее, что делает наша функция — возвращает сформированную структуру Out. Как и в языке C++, это делается с помощью оператора return:

```
return Out;
```

Теперь у нас есть еще одна функция с ключевым словом technique:

```
technique Transform {
    pass P0 {
        VertexShader = compile vs_1_1 ShaderFunc();
    }
}
```

Функция technique позволяет задать технику отображения. Именно ее мы будем вызывать из программы на C++. Внутри такой функции определяются шаги отображения pass. В данном случае только один шаг, который выполняет строку:

```
VertexShader = compile vs_1_1 ShaderFunc();
```

Чтобы лучше понять смысл выполняемых этой строкой действий, попробую проговорить русским языком, что здесь происходит: вершинному буферу присвоить результат компиляции вершинного шейдера версии 1.1 из функции ShaderFunc.

Теперь то же самое, только смотрим на код. Переменная VertexShader указывает на установленный сейчас вершинный шейдер. Ему мы присваиваем результат выполнения оператора compile (компиляция). Далее указывается версия шейдера. Это делается в формате vs_n_m. Значения n и m определяют версию. Чем выше версия, тем больше у него возможностей и больше инструкций он может выполнить, но и тем меньше вероятность, что данная версия будет поддерживаться видеокартой. Поэтому не стоит выбирать последнюю версию, иначе программа может не запуститься на вашем компьютере. Для данного примера нам достаточно будет версии 1.1. Последнее, что нужно указать — функцию с шейдером, которую надо откомпилировать и установить в качестве вершинного шейдера в переменную VertexShader.

Сразу же заметим, что пиксельный шейдер устанавливается примерно таким же образом, только его нужно записать в переменную PixelShader. Например:

```
PixelShader = compile ps_1_1 ShaderFunc();
```

Обратите внимание, что версия пиксельного шейдера указывается как в формате ps_n_m.

1.8.2. Использование вершинного шейдера

Для использования эффекта нам необходим интерфейс `ID3DXEffect`, мы будем подразумевать, что переменная `pEffect` имеет указанный тип:

```
ID3DXEffect* pEffect;
```

Но это еще не все, нам нужно описать вершину, а для этого используется интерфейс `IDirect3DVertexDeclaration9`, поэтому введем еще следующую глобальную переменную:

```
IDirect3DVertexDeclaration9 *VertDecl;
```

Теперь нужно проинициализировать новые переменные. Мы же не зря их добавили в программу. После инициализации и заполнения вершинного и индексного буферов пишем следующий код:

```
// Создаем эффект (шейдер) из файла
D3DXCreateEffectFromFile(pD3DDevice, "simple.vsh", 0, 0, 0, 0,
     &pEffect, 0);
pEffect->SetTechnique("Transform");

// Структура объявления вершин
D3DVERTEXELEMENT9 decl[]= {
  {0, 0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
       D3DDECLUSAGE_POSITION, 0},
   D3DDECL_END()
};
// Создаем объявление вершин
pD3DDevice->CreateVertexDeclaration(decl, &VertDecl);

// Создаем переменную матрицы проекции
float Aspect = (float)iWidth / (float)iHeight;
D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI/4.0f,
         Aspect, 0.1f, 1000.0f);
```

В самом начале мы создаем шейдер с помощью функции `D3DXCreateEffectFromFile`, которая очень похожа на `D3DXCreateEffect`. Загляните сейчас в файл помощи по DirectX 9 и ознакомьтесь с общим видом и параметрами этой функции. Она претерпела серьезные изменения по сравнению с 8-й версией DirectX, и для нее значительно увеличилось количество передаваемых параметров. Минимально необходимо (и именно так мы поступим) указать три параметра:

- `pDevice` — указатель на Direct3D-устройство;
- `pSrcFile` — строку, которая содержит имя загружаемого файла;

- ppEffect — указатель на указатель интерфейса эффекта, куда будет записан результат.

Эффект загружен, но в нем может быть несколько функций типа technique. Следует выбрать ту, которую нужно использовать в данный момент. Поскольку у нас только одна функция technique, то можно сделать это на этапе инициализации с помощью метода SetTechnique, интерфейса ID3DXEffect:

```
pEffect->SetTechnique("Transform");
```

Методу передается имя функции. У нас она называется Transform и именно это имя мы передаем.

После этого объявляется переменная массива decl типа D3DVERTEXELEMENT9. Эта структура необходима для того, чтобы сообщить Direct3D, какие данные будут передаваться в шейдер. В массиве может быть несколько элементов, и каждый будет описывать один компонент данных вершин. В нашем случае допустим, что вершина описывается следующим образом:

```
D3DVERTEXELEMENT9 decl[]= {
  {0, 0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 0},
   D3DDECL_END()
};
```

Здесь объявлен массив значений типа D3DVERTEXELEMENT9. Внутри описания в фигурных скобках через запятую мы должны описать параметры структуры. В нашем случае только один параметр:

```
{0, 0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 0}
```

Здесь в фигурных скобках перечислены значения элементов структуры D3DVERTEXELEMENT9. Описание должно заканчиваться оператором D3DDECL_END().

Структура D3DVERTEXELEMENT9 в общем виде выглядит следующим образом:

```
typedef struct _D3DVERTEXELEMENT9 {
    BYTE Stream;
    BYTE Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

Давайте рассмотрим параметры этой структуры, потому что их понимание необходимо для дальнейшего изучения эффектов.

Stream — номер потока данных. Шейдер может получать данные из нескольких потоков, но мы пока что воспользуемся только одним, поэтому этот параметр сейчас равен нулю;

Offset — смещение в буфере, где хранятся данные. В нашем случае только одна запись. Если бы их было несколько, то в этом параметре необходимо было бы указать смещение от начала. Например, если бы вершина описывалась координатами и нормалью, то описание вершины выглядело бы следующим образом:

```
D3DVERTEXELEMENT9 decl[] = {
  {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_POSITION, 0},
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_NORMAL, 0},
  D3DDECL_END()
};
```

У первой записи второй параметр равен нулю. Это значит, что в вершинном буфере сначала идет позиция вершины, которая описывается тремя значениями типа float (4 байта). Получается, что описание позиции занимает 12 байт (3 значения координаты, умноженные на размер типа float, который описывает каждую координату). Вторая строка описывает нормаль, которая смещена в буфере вершин на значение позиции вершины, поэтому второй параметр равен 12.

Type — тип данных, используемый в описании. У нас координаты вершины описываются типом float, а значит, здесь необходимо указать флаг D3DDECLTYPE_FLOAT3.

Method — метод обработки данных. Мы будем применять значение по умолчанию и флаг D3DDECLMETHOD_DEFAULT.

Usage — для чего будут использоваться данные. В нашем случае данные описывают позицию. Вспомните, как мы описали структуру sVertex. В ней только три координаты вершины, описывающие ее позицию, и ничего больше. Именно это нужно указать в нашей структуре с помощью флага D3DDECLUSAGE_POSITION. Помимо этого вы можете указывать следующие флаги:

- D3DDECLUSAGE_NORMAL — нормаль;
- D3DDECLUSAGE_TEXCOORD — координата текстуры;
- D3DDECLUSAGE_COLOR — цвет;
- D3DDECLUSAGE_FOG — туман.

Это основные флаги, которые вы будете использовать.

Последний параметр структуры D3DVERTEXELEMENT9 (UsageIndex) позволяет задать индекс использования. Задавая разные индексы, можно указать множественные типы использования. Мы индексы применять не будем, поэтому у нас этот параметр равен нулю.

Теперь у нас есть структура, описывающая вершину, но это только структура. Чтобы передать ее Direct3D, необходимо еще создать интерфейс IDirect3DVertexDeclaration9, переменную такого типа мы уже объявили. Создание интерфейса происходит с помощью метода CreateVertexDeclaration:

```
pD3DDevice->CreateVertexDeclaration(decl, &VertDecl);
```

Первый параметр — это массив структур типа D3DVERTEXELEMENT9, а второй параметр — это переменная, указывающая на интерфейс IDirect3DVertexDeclaration9, который необходимо создать.

И напоследок, в глобальной переменной matProjection создаем матрицу проекции. Она будет точно такая же, как и на этапе инициализации Direct3D, потому что код просто скопирован из функции DX3DInitZ.

Теперь переходим к отображению. Возможный вариант функции отображения показан в листинге 1.6.

Листинг 1.6. Функция отображения куба с помощью шейдера

```
void DrawScene() (
 // Объединяем матрицы проекции, просмотра и мировую
 D3DXMATRIX Full;
 D3DXMatrixMultiply(&Full, (D3DXMATRIX*)&World, (D3DXMATRIX*)&View);
 D3DXMatrixMultiply(&Full, &Full, &matProjection);

 // Устанавливаем переменные шейдера
 pEffect->SetValue("TimeFactor", &ViewAngle, D3DX_DEFAULT);
 pEffect->SetValue("mat", &Full, D3DX_DEFAULT);

 // Указываем шейдеру формат вершин
 pD3DDevice->SetVertexDeclaration(VertDecl);

 // Устанавливаем буфер вершин и индексы
 pD3DDevice->SetStreamSource(0, vBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetIndices(iBuffer);

 // Отображаем объект с использованием шейдера
 UINT uPass;
 pEffect->Begin(&uPass, 0);
```

```
 for (int i=0; i<uPass; i++) {
  pEffect->Pass(i);
  pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                  0, 0, iVertsNum, 0, iPointsNum);
 }
 pEffect->End();
}
```

Давайте посмотрим, что здесь происходит. В самом начале мы изменяем переменную ViewAngle. На ее основе будет определяться угол, на который нужно повернуть куб, и на основе этой переменной будет определяться цвет вершин.

После этого мы рассчитываем матрицу поворота на основе переменной ViewAngle. Результат объединяется с матрицей вида и проекции, которые мы создали на этапе инициализации и сохранили в глобальных переменных matProjection и View.

Матрицу мы рассчитали, но устанавливать с помощью метода SetTransform не будем. Почему? Вспомните, что у нас происходит в шейдере. Каждая вершина будет устанавливаться на основе переданной в шейдер матрицы, поэтому текущие настройки проекции не повлияют на положение вершин.

Теперь нам необходимо передать шейдеру значения переменных. Это выполняется с помощью метода SetValue, которому передается три значения:

- строка, которая содержит имя переменной в шейдере. У нас две переменных: TimeFactor и mat;
- значение переменной, которую необходимо установить;
- размер устанавливаемых данных. Можно указать значение D3DX_DEFAULT, и тогда размер данных будет определен по размеру переменной, указанной во втором параметре.

Следующий код показывает, как мы должны задать значения переменных нашего простого шейдера:

```
pEffect->SetValue("TimeFactor", &ViewAngle, D3DX_DEFAULT);
pEffect->SetValue("mat", &Full, D3DX_DEFAULT);
```

Теперь необходимо указать Direct3D, как описаны вершины, передаваемые в шейдер. Когда мы работали без шейдера, то мы использовали метод SetFVF, которому передаются флаги, например:

```
pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
```

Но в случае с шейдерами такая песня не проходит. Тут необходимо указать интерфейс типа `IDirect3DVertexDeclaration9`. Интерфейс у нас есть, а указать его можно с помощью метода `SetVertexDeclaration`:

```
pD3DDevice->SetVertexDeclaration(VertDecl);
```

У метода всего один параметр — устанавливаемый интерфейс, в котором создано описание вершины. Все это у нас уже проделано.

Теперь можно устанавливать буфер вершин и индексный буфер. Тут ничего нового и все мы это уже делали:

```
pD3DDevice->SetStreamSource(0, vBuffer, 0, sizeof(sVertex));
pD3DDevice->SetIndices(iBuffer);
```

Самое последнее, что мы выполним:

```
UINT uPass;
pEffect->Begin(&uPass, 0);
for (int i=0; i<uPass; i++) {
 pEffect->Pass(i);
 pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
        iVertsNum, 0, iPointsNum);
}
pEffect->End();
```

На первый взгляд может показаться, что код получился сложным. Но ничего сложного в нем нет.

1.8.3. Пиксельный шейдер

Вершинные шейдеры хороши, но не всесильны. Например, с их помощью невозможно сделать качественное освещение. Посмотрите на рис. 1.8, где сфера освещается справа. Поверхность стала неравномерной.

Почему цвет сферы получился таким резанным на границе освещенной и неосвещенной поверхности сферы? Чтобы увидеть это, посмотрим на рис. 1.9. Здесь я создал сферу в виде сетки в 3D-редакторе, а потом с помощью простого растрового редактора попытался закрасить правую часть. При этом жирная вертикальная линия показывает границу освещения. При вершинном отображении сцены движок Direct3D оперирует вершинами и гранями, поэтому невозможно сделать плавный переход.

Проблему можно решить двумя способами:

1. Увеличить количество вершин, чтобы сфера создавалась из большего количества треугольников. В этом случае треугольники будут достаточно маленькими, и прерывистость края будет не такой заметной. Но проблема

не решается полностью, к тому же увеличение вершин приводит к тому, что заметно падает производительность. Каждая вершина — это проблема для видеопроцессора.

Рис. 1.8. Освещение сферы сбоку

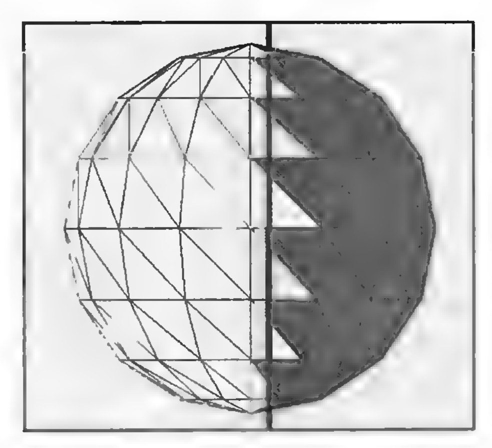


Рис. 1.9. Закрашивание поверхности сферы по определенной границе

2. Использовать пиксельный шейдер. Необходимо установить функцию пиксельного шейдера, и она будет вызываться видеокартой перед отображением каждой очередной точки. Наша задача в этой функции определить цвет каждого пиксела.

Первое решение проблемы вы можете реализовать сами, но я бы не стал этого делать, если есть пиксельный шейдер, который все сделает намного лучше. Это только на первый взгляд кажется, что определение цвета слишком сложная задача, вот сейчас вы увидите, что проблема решается достаточно легко, быстро и непринужденно.

Сначала рассмотрим код шейдера, а потом поговорим о том, как создать пример, использующий его. Код шейдера показан в листинге 1.7.

Листинг 1.7. Код шейдера создания качественного освещения

```
// Глобальные переменные, которые будут задаваться из программы
float4x4 mat : WORLDVIEWPROJECTION;
float4x4 worldmat : WORLD;
float4 light;
float4 facecolor;

// Описание структуры
struct VS_OUTPUT {
    float4 Position : POSITION;
    float3 Normal : TEXCOORD1;
    float3 LightDir : TEXCOORD0;
};

// Функция вершинного шейдера
VS_OUTPUT Vexel_Sh(float4 pos : POSITION, float3 nor : NORMAL){
 VS_OUTPUT Out = (VS_OUTPUT)0;

 // Определяем позицию
 Out.Position = mul(pos, mat);
 // Рассчитываем нормаль
 Out.Normal = normalize(mul(nor, worldmat));
 // Задаем освещение
 Out.LightDir = light;
// Возвращаем результат расчетов
 return Out;
}

float4 Pixel_Sh(float3 normal : TEXCOORD1,
      float3 lightdir : TEXCOORD0): COLOR0 {
```

```
 // Определяем нормаль для освещения
 float3 lightn = normalize(lightdir);
 // Яркость точки равна точки пересечения источника
 // света и нормали света
 float4 diffuse = saturate(dot(lightn, normalized));
 // Результирующий цвет пиксела равен произведению
 // цвета точки на коэффициент
 return facecolor * diffuse;
}

technique PixelLight {
    pass P0 {
        VertexShader = compile vs_1_1 Vexel_Sh();
        PixelShader  = compile ps_2_0 Pixel_Sh();
} }
```

Это самый простейший алгоритм расчета освещения через пиксельный шейдер. Можно было бы реализовать что-то более сложное, но в учебных целях я не стал усложнять жизнь.

Начнем рассмотрение кода с самого конца, а именно с функции PixelLight. Здесь задается не только вершинный шейдер (его роль будет выполнять функция Vexel_Sh), но и пиксельный (это будет функция PixelLight). Обратите внимание, что для пиксельного шейдера используется версия 2.0. Это важно, потому что версия 1.1 не сможет откомпилировать этот код и наверняка вернет ошибку. Для вершинного шейдера мы ничего сложного не используем, поэтому будет достаточно версии 1.1.

Теперь посмотрим на структуру, которая должна описывать выходные данные для вершинного шейдера:

```
struct VS_OUTPUT {
    float4 Position : POSITION;
    float3 Normal : TEXCOORD1;
    float3 LightDir : TEXCOORD0;};
```

Здесь имеется три параметра — позиция вершины, нормаль и позиция освещения. При этом только первые два параметра будут передаваться в вершинный буфер, а третий параметр — будет задаваться. В него мы будем записывать положение источника света. Зачем? Это на практике покажет нам, что при использовании вершинного и пиксельного шейдера одновременно эта структура выполняет роль средства передачи параметров из вершинного в пиксельный шейдер. Положение источника света не изменяется для всей сцены, ведь он находится в одном и том же месте. Но для каждой вершины мы

будем указывать его, чтобы передать это значение в пиксельный шейдер. В реальной программе можно было бы обойтись и без этой переменной.

На уровне вершин мы всего лишь подготавливаем переменные. В качестве входящих данных получаем позицию вершины и ее нормаль (в нашем случае это параметры функции Vexel_Sh). В самом начале функции мы объявляем переменную Out типа VS_OUTPUT и заполняем ее значениями:

```
VS_OUTPUT Out = (VS_OUTPUT)0;
```

Теперь заполняем параметры структуры:

```
Out.Position = mul(pos, mat);
Out.Normal = normalize(mul(nor, worldmat));
Out.LightDir = light;
```

В первой строке определяем позицию вершины, с учетом матрицы положения объекта в пространстве. Для этого перемножаем позицию вершины и переменную mat, где у нас хранится объединенная матрица, объединяющая матрица проекции, мировой и просмотра.

Затем нормализуем нормаль вершины с мировой матрицей. На самом деле, нам придется передавать в шейдер не только объединенную матрицу mat, но и отдельно мировую матрицу.

В последней строке заполняется поле LightDir, куда записываем положение источника освещения, который передается в шейдер через переменную light.

Теперь посмотрим, как будет выполняться формирование сцены. Сначала программа запрашивает вывод определенного объекта. Для каждой вершины объекта вызывается функция вершинного шейдера, где у нас формируется позиция вершины с учетом матриц, и заполняются параметры Normal и LightDir, которые пока никак не влияют на результирующую сцену.

Когда все вершины просчитаны, и положение объекта определено, прежде чем вывести каждый пиксел результирующего объекта, вызывается функция пиксельного шейдера. Ей передаются параметры — нормаль и положение источника света, т. е. второй и третий параметр структуры VS_OUTPUT.

В качестве результата функция пиксельного шейдера возвращает цвет пиксела. Об этом говорит ключевое слово COLOR0, после объявления функции:

```
float4 Pixel_Sh(float3 normal : TEXCOORD1,
    float3 lightdir : TEXCOORD0) : COLOR0
```

Теперь смотрим код функции:

```
float3 lightn = normalize(lightdir);
float4 diffuse = saturate(dot(lightn, normal));

return facecolor * diffuse;
```

Можно в этом коде вместо переменной lightdir использовать напрямую глобальную переменную light, и в этом случае можно убирать соответствующий параметр у функции и поле LightDir у структуры VS_OUTPUT, потому что значение этого поля и переменной light одинаковы.

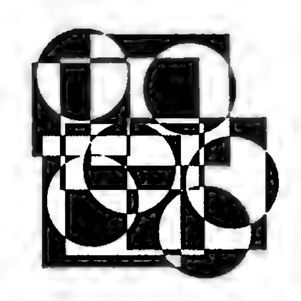
Код расчета цвета пиксела прост. Необходимо определить нормаль для источника освещения. Затем выясняется точка нормали света и нормали, полученной в качестве параметра функции (вспомните, что там находится). Теперь насыщенность пиксела можно определить с помощью функции saturate. Результат выполнения этой функции записывается в переменную diffuse.

Теперь для определения цвета пиксела достаточно перемножить цвет объекта на насыщенность из переменной diffuse. Цвет объекта неизменен и находится в глобальной переменной facecolor.

Как видите, создать качественное освещение с помощью шейдера не так уж и сложно. Не на много сложнее, чем с помощью средств Direct3D. При этом вы можете запрограммировать любой алгоритм освещения, а ведь их много. Каждый из алгоритмов отличается сложностью, качеством и результатом. Например, некоторые объекты должны содержать блики при освещении. К таким объектам относятся зеркальные поверхности, например, фарфоровая посуда. Получается эффект глянцевой поверхности. Другие поверхности должны при освещении выглядеть матовыми и это совершенно другой метод освещения.

В зависимости от поверхности и от источника освещения вы можете применять тот или иной алгоритм освещения. С помощью шейдеров вы легко можете подменять алгоритмы, просто изменяя файл шейдера без перекомпиляции всей программы. Это очень удобно.

ГЛАВА 2



Разработка движка

С основами мы разобрались и пора переходить к разработке игрового движка. В этой области множество различных решений одних и тех же задач. В графике вообще практически любую задачу можно решить сотней способов, и каждый способ по-своему и для своей задачи хорош. Лишь бы он решал поставленную задачу, был удобен программисту и работал быстро.

Прежде чем писать эту книгу, я просмотрел множество исходных кодов игр и изучил немало различных решений, но ни одно из них нельзя назвать идеальным. Некоторые решения направлены на производительность, но удобство сопровождения такого кода оставляет желать лучшего. Некоторые направляют усилие на удобство и расширяемость, но при этом теряют в производительности. А некоторые стремятся получить и то и другое и иногда получают абсурдные решения, которые проигрывают в скорости даже Windows GDI и неповоротливы, как слон. Конечно, мне не были доступны движки, которые являются собственностью компаний и закрыты от посторонних глаз.

Я не претендую на написание лучшего в мире движка игры и не утверждаю, что именно моя реализация лучше. В данной книге движок написан так, чтобы его проще было читать и понимать, и при этом там, где необходимо, даны рекомендации по улучшению и оптимизации. Только опыт позволит вам создать идеальное сочетание скорости и удобства кода. Да и невозможно написать движок, который будет одинаково эффективен для всех игр. Все зависит от жанра, сюжета и самой игры.

Для тех, кто еще даже не пытался создавать собственные игры, данная глава будет интересна с точки зрения определения, как должен выглядеть движок, и рассмотреть возможную структуру. Те, кто уже пытался создавать игры и имеет какие-то наработки, могут воспользоваться определенными идеями или целыми функциями, которые накопились в моей голове.

Я хоть и не писал никогда коммерческих работ, но в данной сфере "кручусь" уже достаточно долго (где-то с 94-го года) и постоянно слежу за современными возможностями и тенденциями развития игрового рынка. Уж очень интересует меня эта сфера и люблю играть с графикой. Возможно, что в будущем и напишу полноценную игру, а не заготовку, и для этого я сейчас изучаю графические пакеты, чтобы научиться создавать 3D-модели.

2.1. Структура движка

В *главе 1* мы рассмотрели базовое приложение, которое создает окно и обрабатывает сообщения. Но это далеко еще не двигатель игры. Помимо обработки сообщений от пользователя необходимо научить программу еще очень и очень многому:

- получению информации от пользователя (ввод с клавиатуры, мыши или джойстика);
- перемещению персонажа по игровому пространству;
- искусственному интеллекту, моделированию действий противников;
- визуализации.

Это основные моменты, которые присутствуют в большинстве игр. Трудно себе представить игру, в которой чего-то из этого не будет. Если основу игры, которую мы создали, желательно создавать без использования объектов (для повышения производительности), то в движке лучше пожертвовать парой тактов процессора ради удобства кода, что позволит упростить сопровождение и расширение игры.

Помимо класса двигателя игры нам понадобится класс для управления персонажами или объектами игры. Все, что должно загружаться из сетки, будет сохраняться в экземпляре такого класса, который должен быть абсолютно автономным и выполнять за нас следующие действия:

- загружать и хранить сетку из X-файла;
- хранить ссылки на эффекты;
- хранить информацию о положении объекта;
- отображать объект.

Основной класс двигателя игры должен создавать экземпляры классов для всех объектов игры, инициализировать необходимые переменные и отображать сцену.

Давайте попробуем создать двигатель, который будет отображать комнату и человека в ней с учетом освещения, чтобы сразу делать сцену реалистичной.

Освещение объекта и тени будут создаваться с помощью шейдеров. Благодаря им, двигатель игры значительно упрощается.

2.2. Двигатель объекта

Для управления объектами игры создадим файлы DXGObject.cpp и DXGObject.h, которые будут хранить класс `CDXGObject`. Заголовочный файл можно увидеть в листинге 2.1.

Листинг 2.1. Описание класса объекта игры

```
#include <string>
#include "d3d9.h"
#include "d3dx9.h"

const int MAX_OBJECT_EFFECTS = 10;

class CDXGObject
{
 public:
  // Сохраним указатель на устройство Direct3D
  IDirect3DDevice9 *pDevice;

  // Информация о сетке
  DWORD dwNumMaterials;
  ID3DXMesh *pMesh;
  LPDIRECT3DTEXTURE9 *pMeshTextures;
  D3DMATERIAL9 *pMeshMaterials;

  // Эффект
  ID3DXEffect *pEffect[MAX_OBJECT_EFFECTS];
  TCHAR *pTechniqueName[MAX_OBJECT_EFFECTS];

  // Источники освещения
  D3DXVECTOR4 Light[MAX_OBJECT_EFFECTS];

  // Матрица положения объекта
  D3DXMATRIX matWorld;
public:
  CDXGObject(IDirect3DDevice9 *pD3DDevice);
  void LoadMeshFromFile(char* filename, char* texturename);
  void SetEffect(ID3DXEffect *effect, int index);
```

```
  void SetTechnique(char* tech, int index);
  const D3DXMATRIX& GetWorldPos() const { return matWorld; };
  void SetWorldPos(D3DMATRIX *matrix);
  void SetLightPos(D3DXVECTOR4 *Pos, int index);
  void Positioning();
  bool Render(DWORD ef_index=1);
  void SetShaderParam(int ef_index);
};
```

О назначении переменных класса можно понять по комментариям. Тут у нас есть все необходимое для хранения следующих данных:

- устройство Direct3D;
- сетка Mesh объекта, используемая в сетке материалов и текстурах;
- эффекты;
- положение источников света, которые могут освещать объект;
- положение объекта.

Константа MAX_OBJECT_EFFECTS определяет максимальное количество эффектов у одного объекта, а также количество объектов освещения. Да, может понадобиться, чтобы один объект выводить несколькими эффектами для разных ситуаций. К тому же, таким образом можно добиться разнообразия. Например, если одну и ту же сетку (объект Mesh) выводить разными эффектами, которые будут окрашивать объект по-разному, или даже немного изменять его форму, то с минимальными усилиями мы получим большое разнообразие объектов из одной сетки.

Но разнообразие — не единственная необходимость использовать несколько шейдеров, и об этом мы узнаем уже из *разд. 2.5*, когда поговорим о тенях.

А зачем каждому объекту нужно знать о количестве источников освещения и их положении? Логичный вопрос, и на него вас ждет простой ответ — освещение будет рассчитываться с помощью шейдеров. Мы будем использовать все современные технологии, и шейдеры относятся к этим и позволят нам сделать максимально реалистичное освещение.

Поскольку класс объекта автономен и должен самостоятельно прорисовывать сетку, то он должен знать о количестве и положении всех источников освещения, которые могут воздействовать на этот объект. Пока будем использовать только один источник, лучи которого подобны солнцу, и будем считать, что они бесконечны. Поэтому у объекта нет информации о длине луча и направлении освещения (это необходимо прожектору), только положение лампочки.

Теперь посмотрим, что представляют и для чего нам нужны функции, которые описаны в заголовочном файле.

- `CDXGObject(IDirect3DDevice9 *pD3DDevice)` — конструктор. В качестве параметра он получает указатель на устройство Direct3D, чтобы сохранить указатель локально для будущего использования.
- `LoadMeshFromFile (char* filename, char* texturename)` — загружает объект из файла. Для загрузки сетки может использоваться функция `LoadMesh`, о которой мы говорили в *разд. 1.5*. В качестве параметров функция получает следующее:
  - имя файла, из которого нужно загрузить шейдер;
  - имя текстуры по умолчанию, которую необходимо загрузить, если она не установлена.
- `SetEffect(ID3DXEffect *effect, int index)` — связывает объект с интерфейсом `ID3DXEffect`. Первый параметр указывает на уже существующий интерфейс `ID3DXEffect` с загруженным шейдером. Второй параметр определяет индекс в массиве интерфейсов, под которым нужно сохранить указатель на интерфейс.
- `SetTechnique(char* tech, int index)` — устанавливает технику отображения `tech` для эффекта с индексом `index`.
- `GetWorldPos()` — возвращает текущую позицию объекта, т. е. мировую матрицу объекта `matWorld`.
- `SetWorldPos(D3DMATRIX *matrix)` — устанавливает мировую матрицу положения объекта, которую передают в качестве параметра.
- `SetLightPos(D3DXVECTOR4 *Pos, int index)` — устанавливает позицию источника освещения с индексом `index` в позицию `Pos`.
- `Positioning()` — устанавливает мировую матрицу положения объекта.
- `Render(DWORD ef_index=1)` — отображает объект с использованием указанного в качестве параметра эффекта.
- `SetShaderParam(int ef_index)` — устанавливает параметры шейдера. В качестве параметра передается индекс самого шейдера.

Реализовать все это можно по-разному. Один из возможных вариантов реализации объекта можно увидеть в листинге 2.2.

Листинг 2.2. Содержимое файла DXGObject.cpp

```
#include "DXGObject.h"
#include "d3d9.h"
```

```
#include "d3dx9.h"
#include "..\..\common\dxfunc.h"

// Конструктор
CDXGObject::CDXGObject(IDirect3DDevice9 *pD3DDevice)
{
 for (int i=0; i<MAX_OBJECT_EFFECTS; i++)
  pEffect[i]=NULL;
 pDevice=pD3DDevice;
 D3DXMatrixIdentity(&matWorld);
}

// Установить эффект
void CDXGObject::SetEffect(ID3DXEffect *effect, int index)
{
 pEffect[index] = effect;
}

// Установить имя техники отображения
void CDXGObject::SetTechnique(char* tech, int index)
{
 if (tech!=NULL)
 {
  pTechniqueName[index] = tech;
  pEffect[index]->SetTechnique(tech);
 }
}

// Установить источник освещения
void CDXGObject::SetLightPos(D3DXVECTOR4 *Pos, int index)
{
 Light[index] = *Pos;
}

// Установить позицию
void CDXGObject::SetWorldPos(D3DMATRIX* matrix)
{
 matWorld = *matrix;
}

// Позиционировать объект в виртуальном мире
// (т. е. установить мировую матрицу)
void CDXGObject::Positioning()
```

```
{
 pDevice->SetTransform(D3DTS_WORLD, &GetWorldPos());
}

// Установить основные параметры шейдера
void CDXGObject::SetShaderParam(int ef_index)
{
 D3DXMATRIX Full, View, matProjection, ident;
 pDevice->GetTransform(D3DTS_VIEW, &View);
 pDevice->GetTransform(D3DTS_PROJECTION, &matProjection);

 D3DXMatrixMultiply(&Full, &matWorld, &View);
 D3DXMatrixMultiply(&Full, &Full, &matProjection);

 if (pEffect[ef_index]->IsParameterUsed("world_matrix",
          pTechniqueName[ef_index]))
   pEffect[ef_index]->SetValue("world_matrix", &matWorld, D3DX_DEFAULT);

 if (pEffect[ef_index]->IsParameterUsed("view_matrix",
          pTechniqueName[ef_index]))
   pEffect[ef_index]->SetValue("view_matrix", &View, D3DX_DEFAULT);

 if (pEffect[ef_index]->IsParameterUsed("proj_matrix",
          pTechniqueName[ef_index]))
  pEffect[ef_index]->SetValue("proj_matrix", &matProjection,
          D3DX_DEFAULT);

 if (pEffect[ef_index]->IsParameterUsed("world_view_proj_matrix",
          pTechniqueName[ef_index]))
  pEffect[ef_index]->SetValue("world_view_proj_matrix", &Full,
          D3DX_DEFAULT);

 pEffect[ef_index]->SetValue("lightPos0", Light[0], D3DX_DEFAULT);
 pEffect[ef_index]->SetValue("lightPos1", Light[1], D3DX_DEFAULT);
 pEffect[ef_index]->SetValue("lightPos2", Light[2], D3DX_DEFAULT);
 pEffect[ef_index]->SetValue("lightPos3", Light[3], D3DX_DEFAULT);
}

// Отобразить объект
bool CDXGObject::Render(DWORD ef_index)
{
 if (pEffect[ef_index]==NULL)
  return false;
```

```
 Positioning();

 SetShaderParam(ef_index);

 UINT uPass;
 pEffect[ef_index]->Begin(&uPass, NULL);

 // Цикл отображения объекта
 for(UINT i = 0; i < uPass; i++)
 {
  pEffect[ef_index]->Pass(i);

  for (DWORD j=0; j<dwNumMaterials; j++)
  {
   pDevice->SetMaterial(&pMeshMaterials[j]);

   // Если есть текстура, то установить ее
   if (pMeshTextures[j])
   {
    pDevice->SetTexture(0, pMeshTextures[j]);
    if (pEffect[ef_index]->IsParameterUsed("tText",
          pTechniqueName[ef_index]))
     pEffect[ef_index]->SetValue("tText", &pMeshTextures[j],
          D3DX_DEFAULT);
   }
   pMesh->DrawSubset(j);
  }
 }
 pEffect[ef_index]->End();

 return true;
}

// Загрузить объект из сетки
void CDXGObject::LoadMeshFromFile(char* filename, char* texturename)
{
  dwNumMaterials = LoadMesh(filename, pDevice,
     &pMesh, &pMeshTextures, texturename, &pMeshMaterials);
}
```

Листинг получился достаточно большим. С некоторыми функциями все понятно, а некоторые требуют дополнительных пояснений. Давайте посмотрим, как происходит отображение объекта. Чтобы его отобразить, необходимо вы-

звать метод Render. В качестве параметра метод получает индекс эффекта, который должен использоваться для отображения.

В самом начале метода проверяем, если эффект с указанным индексом не существует, то прерываем отображение. Отображать объекты будем только через шейдер, поэтому при его отсутствии ничего делать не нужно.

После этого вызываем метод Positioning, который устанавливает мировую матрицу объекта. Следующим этапом вызывается метод SetShaderParam. В этом методе содержится множество конструкций типа:

```
if (pEffect[ef_index]->IsParameterUsed("параметр", техника))
 pEffect[ef_index]->SetValue("параметр", &значение, D3DX_DEFAULT);
```

В первой строке с помощью метода IsParameterUsed проверяем, существует ли в шейдере указанный в качестве первого значения метода параметр. Если да, то устанавливаем этот параметр.

Таким образом, в функции задается матрица проекции, просмотра, мира, а также различные сочетания этих матриц. После этого устанавливается позиция источников освещения. Вот тут я сэкономил на коде и нахально вызываю метод SetValue без проверки, используется параметр или нет. Ошибки от этого не будет, но как-то не элегантно и не красиво. Лучше все же делать проверку с помощью метода IsParameterUsed.

После заполнения основных параметров шейдера начинаем отображать его. Для этого вызывается цикл:

```
UINT uPass;
pEffect[ef_index]->Begin(&uPass, NULL);

for (UINT i = 0; i < uPass; i++)
{
 pEffect[ef_index]->Pass(i);
 // Отображение сетки
}
```

Так как объекты в нашем случае — это сетки Mesh, то это здесь внутри цикла будет еще один цикл:

```
for (DWORD j=0; j<dwNumMaterials; j++)
{
 // Установить материал
 pDevice->SetMaterial(&pMeshMaterials[j]);

 // Если есть текстура, то установить ее
 if (pMeshTextures[j])
```

```
{
  // Установить текстуру
  pDevice->SetTexture(0, pMeshTextures[j]);

  // Указать текстуру шейдеру
  if (pEffect[ef_index]->IsParameterUsed("tText",
      pTechniqueName[ef_index]))
  pEffect[ef_index]->SetValue("tText", &pMeshTextures[j],
      D3DX_DEFAULT);
 }

 // Отобразить j-й элемент сетки
 pMesh->DrawSubset(j);
}
```

Цикл прост, единственный нюанс — если у сетки есть текстура, то проверяем, есть ли параметр `tText` у шейдера, и если есть, то устанавливаем шейдеру текущую текстуру сетки.

Поскольку при отображении используется два цикла, еще и вложенных, то лучший вариант — это если объект "сетка" будет однородным и будет отображаться за один проход. Да, это сложнее в моделировании, особенно если объект сложный и должен содержать множество несхожих текстур.

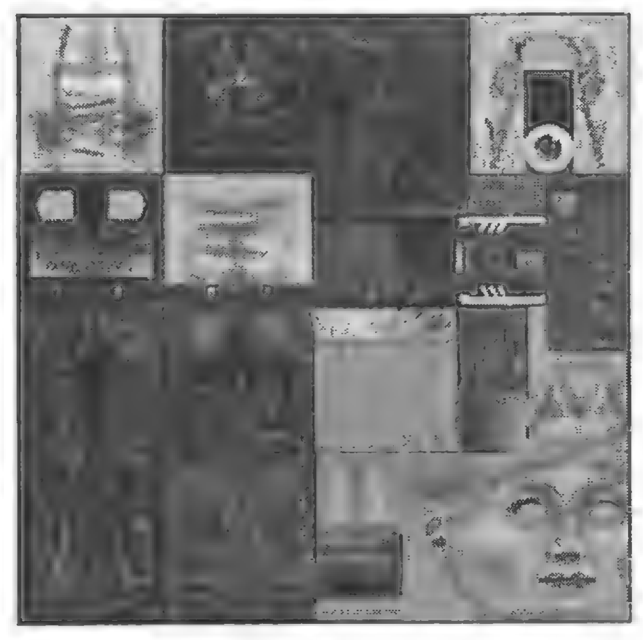


Рис. 2.1. Текстура, которая натягиается на сетку человека из файла tiny.x

На первый взгляд, модель человека нельзя сделать из одной сетки, но это мнение только мнение начинающего программиста. Опытный художник и модельер 3D знает, что это возможно, а профессионал сможет и создать. Яркий пример — сетка человека tiny.x, которую мы уже видели в *разд. 1.6*, и ее текстура Tiny_skin.bmp, которую можно увидеть на рис. 2.1. Именно этот файл натягивается на сетку, образующую человека.

2.3. Двигатель игры

Теперь поговорим о двигателе игры. Чтобы управлять объектами, нам понадобится движок, на плечи которого возложим:

- создание и уничтожение объектов игры;
- отображение сцены.

Пока остановимся на этих двух пунктах. Движения по виртуальному миру, столкновения и искусственный интеллект — это отдельная тема и не стоит с первых же минут усложнять себе жизнь.

Итак, в листинге 2.3 показано описание движка игры. Ознакомьтесь с ним, а затем мы рассмотрим его более подробно.

Листинг 2.3. Описание класса движка игры

```
#include "d3d9.h"
#include "d3dx9.h"
#include "DXGObject.h"

const int NUM_OBJECTS = 20;
const int MAX_EFFECTS = 50;

class CGraphEngine
{
private:
 float ViewAngle;
 ID3DXEffect *pEffect[MAX_EFFECTS];
 TCHAR *pTechniqueName[MAX_EFFECTS];
 IDirect3DDevice9 *pDevice;
 IDirect3DVertexDeclaration9 *VertDecl;
 CDXGObject *objects[NUM_OBJECTS];
public:
 CGraphEngine(IDirect3DDevice9 *pD3DDevice);
 ~CGraphEngine();
 void InitGraphObjects();
```

```
    bool InitObject(int type, int index);
    void LoadEffect(const char* filename, int index);
    void RenderScene();
};
```

Давайте посмотрим, что у нас тут есть и для чего. Для начала разберемся со свойствами класса.

- ViewAngle — угол осмотра. Пример, который мы сейчас создаем, будет вращать камеру для того, чтобы вы могли осмотреть мир с разных сторон.
- pEffect — массив указателей на интерфейсы ID3DXEffect, в которые мы будем загружать шейдеры.
- pTechniqueName — массив имен техник отображений для соответствующих шейдеров из массива pEffect.
- pDevice — указатель на устройство Direct 3D. Для удобства сохраним указатель в переменной класса.
- objects — массив объектов типа CDXGObject, в котором будем хранить объекты сцены.
- VertDecl — указатель на интерфейс IDirect3DVertexDeclaration9 для описания вершины. Это необходимо для работы шейдера.

Теперь посмотрим на методы, которые у нас есть в движке. Помимо конструктора и деструктора у нас есть метод InitGraphObjects, в котором инициализируются объекты сцены. Таких методов в играх может быть несколько для загрузки данных под каждый уровень игры. Для инициализации отдельного объекта вызываем метод InitObject.

Метод LoadEffect будет использоваться для загрузки шейдеров из файла под определенным индексом в массиве pEffect. Последний метод, который есть у нашего движка — RenderScene, который отображает все объекты сцены.

Теперь перейдем к реализации простого движка игры. С ним можно ознакомиться в листинге 2.4.

Листинг 2.4. Реализация простого движка

```
#include "GraphEngine.h"

// Константа, определяющая максимальное количество объектов
const int CURRENT_OBJECTS_NUM = 2;

// Конструктор
CGraphEngine::CGraphEngine(IDirect3DDevice9 *pD3DDevice)
```

```
{
 pDevice=pD3DDevice;
 ViewAngle=0;

 // Описание вершины по умолчанию
 D3DVERTEXELEMENT9 decl[]=
 {
  {0,0,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_POSITION,0},
  {0,12,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_NORMAL,0},

{0,24,D3DDECLTYPE_FLOAT2,D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_TEXCOORD,0},
  D3DDECL_END()
 };
 pDevice->CreateVertexDeclaration(decl, &VertDecl);

 InitGraphObjects();
}

// Деструктор
CGraphEngine::~CGraphEngine()
{
 for (int i=0; i<CURRENT_OBJECTS_NUM; i++)
  delete objects[i];
}

// Инициализация объектов
void CGraphEngine::InitGraphObjects()
{
 // Загрузка эффектов, которые мы будем использовать
 LoadEffect("fx\\room.fx", 0);
 LoadEffect("fx\\obj.fx", 1);

 // Инициализация сетки комнаты
 InitObject(SO_DEFAULT_ROOM, 0);
 InitObject(SO_DX_MAN, 1);

 // Позиционирование комнаты
 D3DMATRIX Pos = {
   1, 0, 0, 0,
   0, 1, 0, 0,
   0, 0, 1, 0,
   0, 2, 0, 1,
 };
 objects[0]->SetWorldPos(&Pos);
```

```
 // Позиционирование человека в комнате
 float b=1;
 D3DMATRIX Pos1 = {
   0.1f, 0, 0, 0,
   0, 0.1f, 0, 0,
   0, 0, 0.1f, 0,
   0, 0, 0, 1,
 };
 D3DXMATRIX Rot;
 D3DXMatrixRotationX(&Rot, -1.5f);
 D3DXMatrixMultiply((D3DXMATRIX*)&Pos1, (D3DXMATRIX*)&Pos1, &Rot);
 objects[1]->SetWorldPos(&Pos1);

 // Создать один источник освещения и назначить его объектам
 D3DXVECTOR4 *lp = new D3DXVECTOR4(20, 20, 0, 0);
 objects[0]->SetLightPos(lp, 0);
 objects[1]->SetLightPos(lp, 0);
}

// Инициализация конкретного объекта
bool CGraphEngine::InitObject(int type, int index)
{
 objects[index] = new CDXGObject(pDevice);

 switch(type)
 {
  // Задать параметры комнаты
  case SO_DEFAULT_ROOM:
   objects[index]->LoadMeshFromFile("Media\\room.x", NULL);
   objects[index]->SetEffect(pEffect[0], 0);
   objects[index]->SetTechnique("PixelLight", 0);
   break;
  // Задать параметры человека
  case SO_DX_MAN:
   objects[index]->LoadMeshFromFile("Media\\tiny.x", NULL);
   objects[index]->SetEffect(pEffect[1], 0);
   objects[index]->SetTechnique("PixelLight",  0);
   break;
 }
 return true;
}

// Загрузить эффект
void CGraphEngine::LoadEffect(const char* filename, int index)
```

3 Зак. 543

```
{
 if (filename==NULL)
  return;
 HRESULT hr = D3DXCreateEffectFromFile(
          pDevice, filename, 0, 0, 0, 0, &pEffect[index], 0);
}

// Отобразить сцену
void CGraphEngine::RenderScene()
{
 // Основные настройки отображения
 pDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_SOLID);
 pDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
 pDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);

 // Здесь необходимо повернуть матрицу просмотра
 ...

 pDevice->SetVertexDeclaration(VertDecl);

 // Отобразить объекты
 for (int i=0; i<CURRENT_OBJECTS_NUM; i++)
  objects[i]->Render(0);
}
```

Давайте мимолетно, но с небольшими остановками пробежимся по коду этого простого движка. Начнем с конструктора. Что здесь интересного? А интересное здесь то, что мы создаем описание вершины `IDirect3DvertexDeclaration9`, которое необходимо для отображения объекта с помощью шейдеров. Мы должны сообщить системе из чего состоят вершины. В нашем случае используются сетки, которые содержат информацию о положении точек, нормалях и текстурных координатах. Поэтому мы описываем вершину следующим массивом:

```
D3DVERTEXELEMENT9 decl[]=
{
  {0,0,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
     D3DDECLUSAGE_POSITION,0},
  {0,12,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
     D3DDECLUSAGE_NORMAL,0},
  {0,24,D3DDECLTYPE_FLOAT2,D3DDECLMETHOD_DEFAULT,
     D3DDECLUSAGE_TEXCOORD,0},
   D3DDECL_END()
 };
```

После создания описания вызывается функция InitGraphObjects, в которой происходит загрузка эффектов и инициализация объектов игры. Для первой тестовой сцены нам понадобятся два объекта (комната и человек) и два эффекта для отображения, по эффекту на каждый объект.

Инициализация объектов происходит достаточно интересно. Для этого в методе InitObject сначала инициализируем в массиве objects элемент с индексом, номер которого передан в качестве второго параметра. После этого следует оператор switch. В зависимости от типа объекта (определяется по первому параметру метода) загружаем сетку, назначаем шейдеры и указываем имя техники отображения. Например, при загрузке человека выполняется следующий код:

```
case SO_DX_MAN:
  objects[index]->LoadMeshFromFile("Media\\tiny.x", NULL);
  objects[index]->SetEffect(pEffect[1], 0);
  objects[index]->SetTechnique("PixelLight",  0);
  break;
```

Сначала с помощью метода LoadMeshFromFile мы загружаем сетку объекта. Во второй строке назначаем эффект, и затем указываем технику отображения.

После загрузки сеток объекты нужно расставить в нашем виртуальном пространстве. Для этого устанавливаются мировые матрицы для объектов objects[0] и objects[1]. Напоследок создадим источник освещения и передадим его координаты обоим объектам сцены.

Теперь переходим к рассмотрению метода отображения. В самом начале устанавливаем основные настройки отображения, например, в данном примере устанавливается сглаживание текстур. После этого необходимо настроить видовую матрицу VIEW, но на данном этапе остановимся на простейшем автоматическом вращении камеры вокруг оси Y для осмотра достопримечательностей. Код вращения был опущен для экономии места. Если возникнут проблемы, вы всегда можете увидеть этот код на компакт-диске.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter2\CoolRoom.

После этого устанавливаем формат вершин. Это делаем один раз, потому что для ускорения работы и уменьшения количества операций все объекты будут отображаться в одном формате вершин.

Все готово. Осталось вызвать метод Render для каждого объекта сцены, и он будет отображен на экране.

Когда долго программируешь, то приходится наблюдать за конкурентами или программистами, действующими на одном с вами поле. Так вот, очень часто

встречается код, в котором каждый объект хранит не ссылку на эффект, а полностью загружает этот эффект и использует единолично. Если у каждого объекта используется уникальный шейдер, то такая техника оправдана. Но такая ситуация встречается очень редко и в большинстве случаев связана с неверным программированием. Большинство объектов имеют схожий алгоритм отображения и могут обойтись одним эффектом.

Итак, если каждый объект будет самостоятельно загружать эффект, то это приведет к следующим проблемам:

- излишнему расходу памяти, потому что у каждого объекта свой экземпляр интерфейса `ID3DXEffect`. В нашем случае создается только один экземпляр, который хранится в движке игры, а объекты получают и хранят только ссылки на этот экземпляр;
- неоправданному расходу процессорного времени. Загрузка шейдера — достаточно трудоемкий процесс для процессора. В этот момент шейдер интерпретируется интерфейсом `ID3DXEffect`, проверяется на ошибки и компилируется в шейдер на языке ассемблера шейдеров.

Наш простой движок избавлен от обоих недостатков. Мы использовали очень простое и элегантное решение. Однажды я видел очень интересную и качественную игру (не будем уточнять какую, потому что не люблю делать антирекламу), в которой загрузка каждого участка игры отнимала до 4-х минут. Это просто ужасно. Ведь при этом загружался даже не целый уровень, а только его небольшой участок. Например, заходишь в дом, идет загрузка. Поднялся на второй этаж, отдохни еще пять минут. Прошел мимо пары домов, опять перекур на 2—4 минуты и т. д. Так играть просто невозможно. При этом ведь объекты все одни и те же и шейдеры не изменяются. Налицо отсутствие оптимизации загрузки объектов.

2.4. Запуск движка

Теперь посмотрим, как можно запустить движок в работу. В *главе 1* мы написали базовый модуль и приложение, которые будут использоваться нашей программой. Давайте посмотрим, как к этому базовому приложению достаточно просто "прикрутить" движок. Для этого нужно всего три телодвижения.

Первое движение необходимо для создания глобальной переменной и для хранения указателя на экземпляр движка типа `CGraphEngine`. Например:

```
CGraphEngine* ge;
```

Второе движение необходимо для инициализации движка. Его нужно сделать сразу после инициализации Direct3D:

```
ge = new CGraphEngine(pD3DDevice);
```

И последнее — заставить сцену отображаться. Для этого в цикле обработки сообщения добавьте вызов метода GraphEngine, который может выглядеть следующим образом:

```
void GraphEngine()
{
 if (SUCCEEDED(pD3DDevice->BeginScene()))
 {
  pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
       D3DCOLOR_XRGB(255,255,255), 1.0f, 0);
  ge->RenderScene();
  pD3DDevice->EndScene();
 }

 pD3DDevice->Present(NULL, NULL, NULL, NULL);
}
```

Все достаточно просто:

1. Начинаем формирование сцены с помощью метода BeginScene.
2. Очищаем буфер с помощью метода Clear.
3. Вызываем метод RenderScene, чтобы он вывел все объекты на экран.
4. Завершаем формирования сцены (EndScene).
5. Отображаем результат на экране Present.

Вот и все. На рис. 2.2 можно увидеть результат выполнения программы. Обратите внимание, что сцена получилась достаточно реалистичной. Во время движения вокруг человека можно заметить, что одна сторона освещается больше другой. Дело в том, что с помощью шейдера мы учитываем освещение.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter2\CoolRoom.

Итак, давайте посмотрим, на шейдер который выводит объект. У нас загружается два эффекта из двух файлов: obj.fx и room.fx. На самом деле, в данном примере можно было обойтись и одним файлом с единственным шейдером, и просто назначить его двум объектам. Движок это позволяет, но я использовал два шейдера, потому что в ближайшем будущем объект и комната будут выводиться немного по-разному.

Итак, код шейдера, который находится в файлах obj.fx и room.fx, можно увидеть в листинге 2.5.



Рис. 2.2. Результат отображения человека и комнаты с помощью движка

Листинг 2.5. Код шейдера с учетом освещения

```
float4x4 world_view_proj_matrix;
float4x4 view_proj_matrix;
float4x4 view_matrix;
float3 lightPos0;

// Структура входных данных для вершинного шейдера
struct VS_INPUT1
{
   float3 Pos:      POSITION;
   float3 Normal:   NORMAL;
   float2 TexCoord: TEXCOORD;
};

// Структура выходных данных вершинного шейдера
struct VS_OUTPUT1
```

```
{
   float4 Pos:       POSITION;
   float2 TexCoord: TEXCOORD0;
   float3 LightDir: TEXCOORD1;
   float3 Normal:   TEXCOORD2;
};

// Функция вершинного шейдера
VS_OUTPUT1 Vexel_ShObj(VS_INPUT1 In)
{
   VS_OUTPUT1 Out;

   // Корректируем позицию с учетом матрицы положения
   Out.Pos = mul(float4(In.Pos,1.0), world_view_proj_matrix);

   Out.TexCoord = In.TexCoord; // Задаем текстурную координату
   Out.LightDir = lightPos0;   // Задаем положение цвета
   Out.Normal = In.Normal;     // Задаем нормаль

   return Out;
}

// Объявляем переменную для текстуры
sampler2D tText;

// Выходная структура для пиксельного шейдера
struct PS_OUTPUT
{
    float4 Color : COLOR;
};

// Входная структура для пиксельного шейдера
struct PS_INPUT1
{
   float2 TexCoord: TEXCOORD0;
   float3 LightDir: TEXCOORD1;
   float3 Normal:   TEXCOORD2;
};

// Пиксельный шейдер
PS_OUTPUT Pixel_ShObj( PS_INPUT1 In )
{
   PS_OUTPUT Out;
```

```
    // Определяем цвет вершины точки. Для этого берем цвет
    // пиксела текстуры по координате In.TexCoord
    float4 color  = tex2D(tText, In.TexCoord);

    // Корректируем цвет с учетом освещения
    float3 normal = normalize(In.Normal);
    float3 lightDir = In.LightDir;
    float dotP = saturate(dot(-lightDir, normal));

    // Чтобы не было абсолютно черных частей на объекте,
    // в качестве коэффициента берем максимум между 0.4 и
    // значением dotP. Таким образом, хотя бы на 40% объект
    // будет освещен в любом случае
    dotP = max(dotP, 0.4);

    // Корректируем цвет с учетом коэффициента
    Out.Color = color * dotP;

    return Out;
}

// Техника отображения
technique PixelLight
{
     // Первый и пока единственный шаг отображения
     pass P0
     {
         VertexShader = compile vs_2_0 Vexel_ShObj();
         PixelShader  = compile ps_2_0 Pixel_ShObj();
     }
}
```

Код шейдера получился достаточно большим, но это только на первый взгляд. Большую часть этого кода занимает объявление структур и комментарии. Без структур код усложнится и его будет неудобно читать и сопровождать, а так код получился очень красивым. В сочетании с подробными комментариями вы легко можете разобраться с выполняемыми действиями.

2.5. Тени

Чтобы заставить наш пример, описанный в *разд. 2.4*, отображать тени, не нужно слишком большого количества усилий. Необходимо внести минимум изменений в код и написать пару дополнительных шейдеров, и программа

будет готова. Итак, давайте посмотрим, какие нужны изменения и где, ведь тени позволят сделать нашу комнату максимально реалистичной.

В движке объектов изменения не понадобятся, потому что он очень хороший и в таком виде подходит нам. А вот в движке игры потребуются некоторые модификации. Первое — необходимо загрузить три шейдера для наших двух объектов комнаты и человека:

```
LoadEffect("fx\\room.fx", 0);
LoadEffect("fx\\obj.fx", 1);
LoadEffect("fx\\floor.fx", 2);
```

Здесь добавлена загрузка эффекта с индексом 2, и его мы должны назначить комнате.

Теперь переходим к инициализации объектов. Инициализация человека не изменится, потому что тут просто расширится файл шейдера. А вот у комнаты появился еще один интерфейс эффекта и его нужно назначить:

```
case SO_DEFAULT_ROOM:
  objects[index]->LoadMeshFromFile("Media\\room.x", NULL);
  objects[index]->SetEffect(pEffect[0], 0);
  objects[index]->SetEffect(pEffect[2], 1);
  objects[index]->SetTechnique("PixelLight", 0);
  objects[index]->SetTechnique("PixelLight", 1);
  break;
```

В данном случае, после загрузки сетки мы устанавливаем эффекты с помощью метода `SetEffect`. Сначала эффект с индексом 0 устанавливаем объекту под индексом 0. Потом эффект с индексом 2 устанавливаем объекту под индексом 1. Теперь достаточно только установить техники отображения с помощью метода `SetTechnique`.

И так посмотрим, как изменился код отображения сцены:

```
void CGraphEngine::RenderScene()
{
 // Начальные настройки отображения
 ...

 // Поворот сцены и установка матрицы вида
 ...

 pDevice->SetVertexDeclaration(VertDecl);

 // Отобразить нулевой объект (комнату)
 // с использованием нулевого шейдера
 objects[0]->Render(0);
```

```
 pDevice->Clear(0, NULL, D3DCLEAR_STENCIL,
      D3DCOLOR_XRGB(255,255,255), 1.0f, 0x40);

 // Цикл отображения объектов
 for (int i=1; i<2; i++)
  objects[i]->Render(0);

 // Снова отображаем комнату с использованием первого шейдера
 objects[0]->Render(1);
}
```

Отображение сцены происходит в три этапа. Сначала выводим сетку комнаты. Здесь используется уже знакомый нам шейдер из предыдущего примера, который отображает объект только с учетом освещения.

После этого очищаем Stencil-буфер, который как раз и удобен для построения маски. Для очистки применяется метод `Clear`, который используется для очистки экрана и Z-буфера.

Теперь запускается цикл, который отображает все объекты, которые должны отбрасывать тень на уже отображенную комнату. Вот тут для воспроизведения используется очень интересный шейдер, который не только отображает объект, но и использует Stencil-буфер для подготовки маски, по которой будет строиться тень. Шейдер для объектов можно увидеть в листинге 2.6.

Листинг 2.6. Шейдер отображения объекта и заполнения Stencil-буфера

```
float4x4 world_view_proj_matrix;
float4x4 view_proj_matrix;
float4x4 view_matrix;
float3 lightPos0;
float ExtrudeDistance = 1000;

// Структура входных данных
struct VS_INPUT1
{
   float3 Pos:      POSITION;
   float3 Normal:   NORMAL;
   float2 TexCoord: TEXCOORD;
};

// Структура выходных данных
struct VS_OUTPUT1
{
   float4 Pos:      POSITION;
```

```
   float2 TexCoord: TEXCOORD0;
   float3 LightDir: TEXCOORD1;
   float3 Normal:   TEXCOORD2;
};

// Вершинный шейдер объекта
VS_OUTPUT1 Vexel_ShObj(VS_INPUT1 In)
{
   VS_OUTPUT1 Out;

   Out.Pos = mul(float4(In.Pos,1.0), world_view_proj_matrix);
   Out.TexCoord = In.TexCoord;

   Out.LightDir = lightPos0;
   Out.Normal = In.Normal;

   return Out;
}

sampler2D tText;

// Выходная структура пиксельного шейдера
struct PS_OUTPUT
{
    float4 Color : COLOR;
};

// Входная структура пиксельного шейдера
struct PS_INPUT1
{
   float2 TexCoord: TEXCOORD0;
   float3 LightDir: TEXCOORD1;
   float3 Normal:   TEXCOORD2;
};

// Пиксельный шейдер
PS_OUTPUT Pixel_ShObj(PS_INPUT1 In)
{
  // Расчет цвета пиксела с учетом освещения
}

// Входная структура шейдера тени
struct VS_INPUT
```

```
{
   float3 Pos:         POSITION;
   float3 Normal:      NORMAL;
};

// Выходящая структура шейдера тени
struct VS_OUTPUT
{
   float4 Pos:     POSITION;
   float4 Color:   COLOR;
};

// Вершинный шейдер, рисования лучей тени
VS_OUTPUT Vexel_Sh0( VS_INPUT In )
{
   VS_OUTPUT Out;

   float3 lightDirInView = lightPos0;
   float3 normalInView   = In.Normal;
   float3 posInView      = float4(In.Pos,1.0);

   // Если необходимо, то рисуем луч
   if (dot(normalInView, -lightDirInView)<0.0)
   {
      posInView += lightDirInView * ExtrudeDistance;
   }

   Out.Pos = mul(float4(posInView, 1.0), world_view_proj_matrix);
   Out.Color = float4(0, 0, 0, 0);

   return Out;
}

// Техника отображения
technique PixelLight
{
    pass P0
    {
        VertexShader = compile vs_2_0 Vexel_ShObj();
        PixelShader  = compile ps_2_0 Pixel_ShObj();
    }

    pass P1
    {
      ColorWriteEnable = 0;
```

```
        ZFunc            = Less;
        ZWriteEnable     = false;
        StencilEnable    = True;
        StencilFunc      = Less;
        StencilMask      = 0xffffffff;
        StencilWriteMask = 0xffffffff;

        CullMode = CCW;
        StencilZFail      = IncrSat;

        VertexShader = compile vs_2_0 Vexel_Sh0();
    }

    pass P2
    {
        CullMode = CW;
        StencilZFail      = DecrSat;

        VertexShader = compile vs_2_0 Vexel_Sh0();
    }
}
```

Половина этого листинга вам уже знакома по *разд. 2.4*. Это отображение объекта с учетом выбранной модели освещения. У нас модель пока не изменится, поэтому код функции `Pixel_ShObj` я опустил, чтобы сэкономить место (код этой функции можно взять из листинга 2.5 из *разд. 2.4*).

Начнем рассмотрение шейдера с техники отображения, т. е. с процедуры `technique PixelLight`. В этой процедуре у нас три шага `pass` с именами P0, P1 и P2.

Первый шаг просто отображает объект с учетом освещения. Функции вершинного (`Vexel_ShObj`) и пиксельного (`Pixel_ShObj`) шейдера, которые используются для отображения, уже рассматривались в *разд. 2.4*. Эти функции не изменились и удачно выполняют свою задачу.

Шаги P1 и P2 не выводят графику. Их назначение нарисовать лучи от источника света сквозь объект и уронить эти лучи на стены/пол/потолок или на любой другой объект, который попадется на пути. При этом луч начинает рисоваться от объекта. От источника света берется только направление. Получается, что эти лучи символизируют тень (рис. 2.3).

Примечание

Чтобы увидеть лучи во время выполнения программы, достаточно изменить параметр `ColorWriteEnable` на шаге P1 на значение равное 1.



Рис. 2.3. Лучи создания тени

Теперь посмотрим, какие настройки происходят перед вызовам процедуры вершинного шейдера на шаге P1.

- ColorWriteEnable — устанавливаем в 0, т. е. отключаем запись цвета. У тени цвет нас не особо интересует, потому что нам необходимо только заполнить Stencil-буфер.
- ZFunc — устанавливаем в Less. При определении отдаления объекта (координата Z) используется функция Less.
- ZWriteEnable — устанавливаем в false, т. е. отключаем запись в Z-буфер.
- StencilEnable — устанавливаем в true, т. е. включаем Stencil-буфер.
- StencilFunc — функция Stencil-буфера будет Less, которая выбирает меньшие значения.

Это основные параметры, которые вы должны установить, и они будут использоваться не только на шаге P1, но и на шаге P2. На обоих шагах выполняется одна и та же функция вершинного шейдера Vexel_Sh0. Разница только в том, что P1 устанавливает следующие два параметра:

```
CullMode = CCW;
StencilZFail = IncrSat;
```

А на шаге P2 эти параметры устанавливаются в:

```
CullMode = CW;
StencilZFail = DecrSat;
```

Вот и вся разница между этими двумя шагами. А что делает функция шейдера Vexel_Sh0? Чтобы понять, достаточно посмотреть на следующую проверку:

```
if (dot(normalInView, -lightDirInView)<0.0)
{
 posInView += lightDirInView * ExtrudeDistance;
}
```

Если точка пересечения нормали и освещения меньше нуля, то необходимо провести луч. Для этого текущая позиция увеличивается на значение позиции источника света, умноженное на константу ExtrudeDistance. Значение константы зависит от размеров мира и самой дальней точки в комнате, которая может получать тень. В нашем случае вполне достаточно числа 1000.

После отображения всех объектов и заполнения буфера глубины необходимо снова отобразить комнату, но при этом, используя значения буфера глубины, нарисовать не саму комнату (это мы уже сделали в самом начале), а нарисовать тени. Для этого комната рисуется с использованием шейдера из файла floor.fx, который вы можете увидеть в листинге 2.7.

Листинг 2.7. Использование Stencil-буфера для рисования тени

```
float4x4 world_view_proj_matrix:WORLDVIEWPROJECTION;
float4x4 worldmat : WORLD;
float4 lightPos0;

// Входная структура для вершинного шейдера
struct VS_INPUT
{
   float3 Pos:       POSITION;
   float3 Normal:    NORMAL;
   float2 TexCoord: TEXCOORD;
};

// Выходная структура для вершинного шейдера
struct VS_OUTPUT {
   float4 Pos:       POSITION;
```

```
   float2 TexCoord: TEXCOORD0;
};

// Вершинный шейдер
VS_OUTPUT Vexel_Sh(VS_INPUT In){
   VS_OUTPUT Out;

   Out.Pos = mul(float4(In.Pos,1.0), world_view_proj_matrix);
   Out.TexCoord = In.TexCoord;
   return Out;
}

sampler2D tText;

// Входная структура для пиксельного шейдера
struct PS_INPUT
{
   float4 Pos:      POSITION;
   float2 TexCoord: TEXCOORD;
};

// Пиксельный шейдер
float4 Pixel_Sh(PS_INPUT In): COLOR0{
   float4 color = tex2D(tText,In.TexCoord);
   return color*0.2;
}

// Техника отображения
technique PixelLight
{
  pass P0
  {
    // Включаем альфа-смешивание
    AlphaBlendEnable = True;
    SrcBlend         = DestAlpha;
    DestBlend        = One;

    ZWriteEnable     = true;

    // Настройка Stencil-буфера
    StencilEnable    = True;
    StencilFunc      = Greater;
    StencilRef       = 0x40;
```

```
    // Выполнение шейдера
    VertexShader = compile vs_2_0 Vexel_Sh();
    PixelShader  = compile ps_2_0 Pixel_Sh();
  }
}
```

Снова начнем рассмотрение с техники отображения. В данном случае необходимо включить не только Stencil-буфер, но и альфа-смешивание. Это небольшой трюк, который может сделать тень более естественной.

В вершинном шейдере выполняется всего две строки кода:

```
Out.Pos = mul(float4(In.Pos,1.0), world_view_proj_matrix);
Out.TexCoord = In.TexCoord;
```

В первой мы корректируем позицию вершины с учетом суммарной матрицы положения вершины (мировой, просмотра и проекции). Во второй строке сохраняем текстурные координаты, чтобы воспользоваться ими в пиксельном шейдере.

В пиксельном шейдере все еще более банально:

```
float4 color = tex2D(tText,In.TexCoord);
return color*0.2;
```

В первой строке мы определяем цвет точки из текстуры по текстурной координате, а во второй строке умножаем цвет на коэффициент, чтобы сделать его темнее, как это бывает у теней. На мой взгляд, коэффициент 0.2 дает достаточно эффектный результат. Конечно, это не соответствует физическим законам, но результат красивый и достаточно приемлемый.

Запустите пример, и теперь на стене и полу можно будет наблюдать тень человека. Результат выполнения можно увидеть на рис. 2.4.

Вот так мы получили более реалистичную сцену, в которой появилось не только освещение, но и тень. При этом в код движка были внесены минимальные изменения, и то в тех местах, где определяются загружаемые объекты и шейдеры.

Если еще немного поработать над движком, то можно сделать его расширяемым без компиляции. Достаточно будет только изменять шейдеры и специальный конфигурационный файл, который будет хранить информацию о загружаемых сетках и соответствующих файлах с шейдером.

Мы не будем сейчас рассматривать тему универсального движка и возможные конфигурационные файлы с описаниями, потому что это уже зависит от типа игры и личных предпочтений программиста. Я бы тут порекомендовал использовать файлы в формате XML. Если вы работаете в команде разработ-

чиков, и графику/шейдеры создает специалист в этой области, то ему будет достаточно просто конфигурировать XML-файл и не придется писать специализированную утилиту.

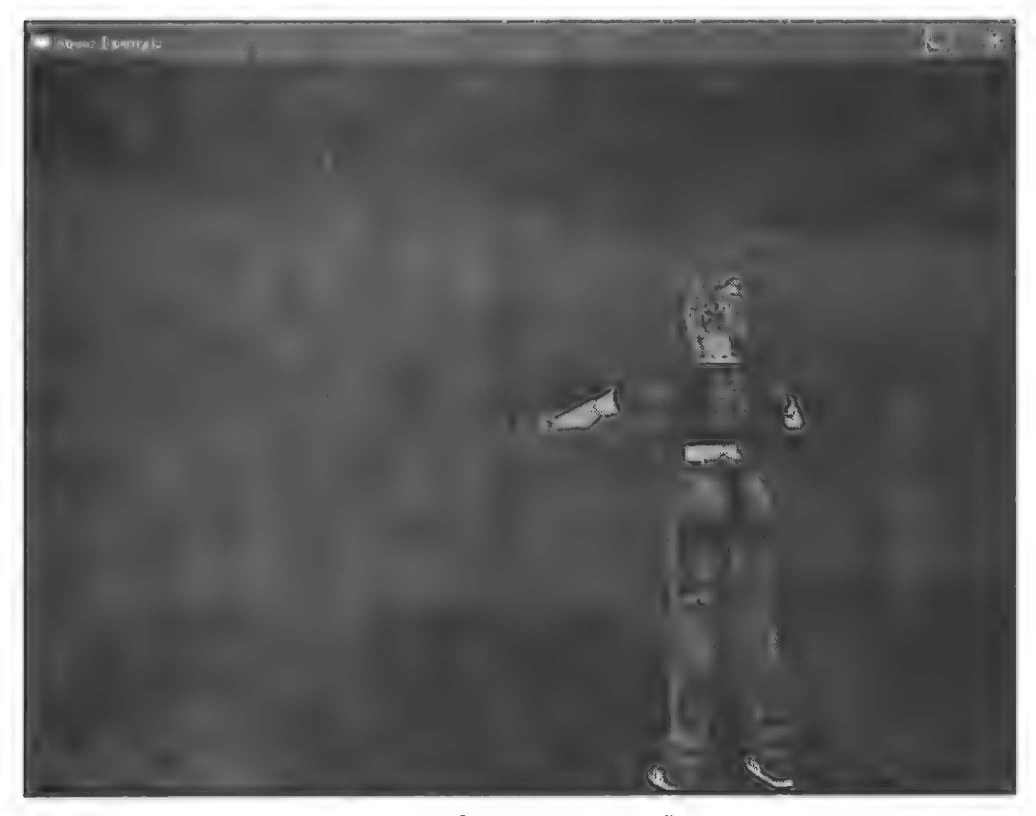


Рис. 2.4. Сцена с учетом теней

Да, разбор XML отнимает достаточно много ресурсов, и бинарный формат с жесткой структурой загружался бы быстрее. Но т. к. загрузка данных происходит при старте программы или при переходах между уровнями, то в этот момент небольшой потерей производительности можно пожертвовать, но только небольшой. Как мы уже говорили в этом разделе, слишком долгая загрузка может раздражать пользователя до такой степени, что он забудет вашу игру, как страшный сон.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter2\Shadow.

2.6. Множество источников освещения

Давайте еще больше усложним задачу — добавим возможность учета нескольких источников освещения. Да, пока что наш пример движка учитывает только одну лампочку на объект, и чтобы исправить этот недостаток, необходимо сделать не так уж и много телодвижений.

Первое, и самое простое — необходимо создать новую лампочку:

```
D3DXVECTOR4 *lp = new D3DXVECTOR4(5, -5, 0, 0);
objects[0]->SetLightPos(lp, 0);
objects[1]->SetLightPos(lp, 0);

D3DXVECTOR4 *lp1 = new D3DXVECTOR4(-5, 5, 0, 0);
objects[0]->SetLightPos(lp1, 1);
objects[1]->SetLightPos(lp1, 1);
```

Сначала создаем вектор расположения первой лампы и устанавливаем его в качестве источника освещения для каждого объекта под индексом 0. Затем создается еще один вектор расположения для второго источника освещения, и его устанавливаем для обоих объектов под индексом 1.

Теперь модифицируем метод Render следующим образом:

```
bool CDXGObject::Render(DWORD ef_index, int light);
{
 int y=dwNumMaterials;
 if (pEffect[ef_index]==NULL)
  return false;

 // Позиционируем объект
 Positioning();

 // Устанавливаем основные параметры
 SetShaderParam(ef_index);

 if (light==-1)
 {
  // Если индекс света равен -1, то установить все
  pEffect[ef_index]->SetValue("lightPos0", Light[0], D3DX_DEFAULT);
  pEffect[ef_index]->SetValue("lightPos1", Light[1], D3DX_DEFAULT);
  pEffect[ef_index]->SetValue("lightPos2", Light[2], D3DX_DEFAULT);
  pEffect[ef_index]->SetValue("lightPos3", Light[3], D3DX_DEFAULT);
  pEffect[ef_index]->SetValue("lightPos4", Light[4], D3DX_DEFAULT);
 }
```

```
 else
 {
  // Если индекс света больше нуля, то использовать указанный
  pEffect[ef_index]->SetValue("lightPos0", Light[light],
    D3DX_DEFAULT);
 }

 // Отобразить объект
 ..
}
```

Теперь метод получает два параметра — индекс эффекта, который нужно использовать, и источник освещения. При этом если второй параметр равен –1, то шейдеру устанавливаются все источники света, иначе только указанный цвет записывается в параметр `lightPos0`.

Теперь о шейдере. У нас объект человека рисовался с помощью одного лишь файла эффекта. Это не эффективно, ведь шаг P0 в шейдере идентичен шейдеру из файла room.fx, который используется для отображения стены. Давайте оптимизируем этот участок кода. Для этого достаточно удалить шаг P0 из эффекта obj.fx. В связи с этим изменится и код отображения сцены:

```
// Отображаем комнату
objects[0]->Render(0);

// Очищаем Stencil-буфер
pDevice->Clear(0, NULL, D3DCLEAR_STENCIL,
    D3DCOLOR_XRGB(255,255,255), 1.0f, 0x40);

// Цикл отображения объектов
for (int i=1; i<2; i++)
{
 // Отображаем сам объект с помощью эффекта 0
 objects[i]->Render(0);

 // Рисуем лучи эффектом 1 для каждого источника света
 objects[i]->Render(1, 0);
 objects[i]->Render(1, 1);
}

// Отображаем тени
objects[0]->Render(1);
```

Изменился цикл отображения объектов. Теперь, сначала отображаем объект с помощью шейдера, который используется и для отображения комнаты

(room.fx). Затем для каждого источника освещения рисуем лучи с помощью эффекта под индексом 1 (там находится шейдер из файла obj.fx).

И последнее, что изменяется — пиксельный шейдер для отображения комнаты. Он будет учитывать сразу два источника освещения, но по тому же принципу вы можете сделать и три, и даже 10 лампочек:

```
// По текстуре определяем цвет вершины
float4 color  = tex2D(tText, In.TexCoord);
float3 normal = normalize(In.Normal);

// Определяем пересечение
float dotP = saturate(dot(-lightPos0, normal));
float dotP1 = saturate(dot(-lightPos1, normal));

// Определяем максимальный коэффициент
// освещения в точке
dotP = max(dotP, dotP1);
dotP = max(dotP, 0.2);

// Умножаем цвет на коэффициент
Out.Color = color * dotP;
```

Сначала определяем цвет и нормализуем нормаль. Здесь, все как и было ранее. После этого определяем коэффициент для каждого источника цвета. Какой из них использовать? Нужно использовать тот, который имеет большую силу, поэтому в качестве результата берем максимальный из двух. Если у вас три источника, то нужно найти наибольший из всех трех.

Чтобы не было абсолютно черной темноты, сравниваем полученный максимум со значением 0.2. Хотя бы на 0.2 единицы освещение должно быть, чтобы объект можно было увидеть и он не утонул в темноте.

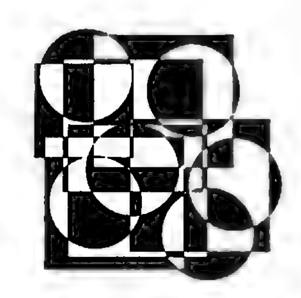
Такое определение освещения является не совсем корректным и немного нарушает законы физики, но зато работает очень быстро и просто. Зная физику света, вы легко сможете подкорректировать пример.

Как видите, не так уж и сложно модифицировать пример, использующий для отображения шейдеры на применение нескольких источников освещения.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter2\Shadow2.

ГЛАВА 3



Скелетная анимация

Скелетная анимация — это не просто дань моде, это удобный, эффективный и достаточно производительный метод создания реалистичной анимации движения людей и животных. Само понятие имеет общие корни с анатомией людей и животных. При движении рукой мы сгибаем локоть и одновременно с этим движутся кости руки. Примерно то же самое происходит и в виртуальном мире при использовании скелетной анимации, только в более упрощенном варианте.

Что упрощено в скелетной анимации? Дело в том, что кости не движутся сами, ими движут мышцы. Мышцы — это сила, которая воздействует на кости, и именно эта сила заставляет кости вращаться. Физику сил мы опустим, но она очень важна. Чем больше вы будете уделять внимания анатомии движения и силам, воздействующим на кости, тем более реалистичную графику вы сможете получить. А для этого придется изучать анатомию и наблюдать за человеческим и животным миром.

Реалистичность движения зависит и от того, как реалистично будет создан скелет, и тут снова все зависит от анатомии. Следите за животным миром. Если в вашей игре будут коты, то последите за тем, как двигается ваш кот, а если нужны лошади, то придется идти на ипподром или даже в зоопарк. Ну а если у вас только роботы и инопланетные монстры, то можете двигать их костями как угодно, просто скажите, что так и должно быть.

Если создать сетку (mesh), максимально приближенную к реальности, и правильно привязать к такой сетке кости, то анимация на основе такого скелета может быть действительно похожа на реальную. Да, это красиво на словах, а когда садишься за клавиатуру, мы сталкиваемся со множеством проблем, которые еще надо решить и о некоторых этих проблемах нам предстоит поговорить.

3.1. Что такое скелет

Все начинается с создания персонажа. С помощью программы 3D-моделирования мы должны создать 3D-модель человека/монстра, которого нужно анимировать. Тут я привык к классике — 3D Studio Max (в простонародье 3ds). Да, эта классика тяжеловесная и стоит дорого, но уж очень удобна и люба сердцу, начиная еще с варианта для MS-DOS.

Сама программа 3D Studio Max позволяет только моделировать сетку персонажа, но это далеко не все. После этого необходимо создать кости и привязать к ним вершины сетки. Двигая кость, все связанные вершины будут также перемещаться и изгибаться. Для создания скелета в 3ds есть модуль — Character Studio. Еще недавно этот модуль был отдельной программой, но сейчас он встроен непосредственно в оболочку в меню **Character**. Для создания реалистичного персонажа знаний только программных пакетов будет маловато. Необходимы хорошие знания анатомии и физики, иначе в реалистичность движений никто не поверит.

Нет, мы не будем сейчас изучать процесс создания моделей и скелетов, потому что это тема отдельной книги. Если вы хотите узнать об этой теме подробнее, то рекомендую прочитать книгу "3ds max 6 и character studio 4. Анимация персонажей" [5]. Отличная книга, но для ее понимания понадобятся начальные знания по самой программе 3ds. Авторы описывают только то, что касается анимации персонажей, и опускают базовые понятия моделирования.

Но зачем же я завел разговор о моделировании, если мы не будем его рассматривать? Как создать персонаж — это отдельная тема, а сейчас нам нужно понимать, что должно быть создано, чтобы уяснить, чем мы будем оперировать в собственной программе.

Итак, давайте посмотрим, из чего состоит скелет. Для иллюстрации примера я взял один из файлов книги "3ds max 6 и character studio 4. Анимация персонажей". На рис. 3.1 вы можете увидеть модель девушки и рядом стоящий ее скелет. Девушка — это то, что слева, а скелет — это справа, по крайней мере, мне так кажется ☺. Скелет 3D-модели похож на настоящий скелет человека. Он состоит из костей, связанных между собой. Если повернуть плечо, то повернется вся рука, вплоть до кончиков пальцев. Поворачивая кисть, поворачиваются и все пальцы.

Вместе с вращением костей движутся и связанные точки сетки персонажа. Это значит, что если повернуть кость, символизирующую плечо, то повернутся все дочерние кости, вплоть до кончиков пальцев, и при этом изменят положения и все связанные точки, какой бы сложной формой они ни были. Изменить положение кости достаточно просто.

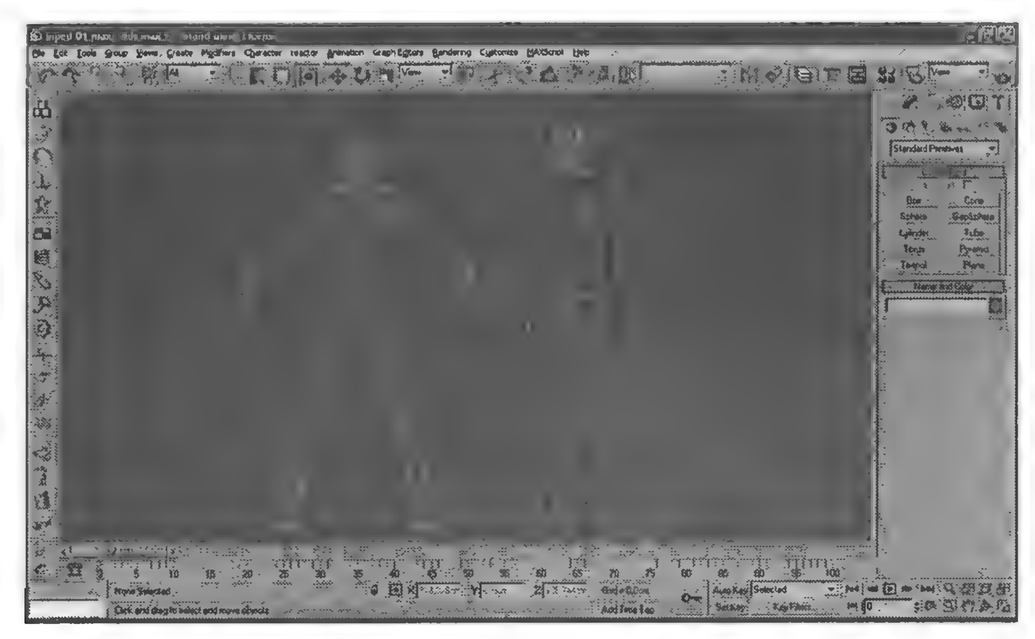


Рис. 3.1. Слева сетка человека похожего на женщину ☺, а справа — скелет

Достаточно только изменить лишь одну матрицу, которая определяет положение кости относительно точки соединения с главной костью. Например, пальцы ноги изменяют положение относительно ступни, ступня относительно голени, голень относительно колена и т. д.

Одна точка сетки может быть привязана к нескольким костям. В этом случае, точка сетки смещается в соответствии с указанным весом точки. Если вершина принадлежит двум костям, то по умолчанию вес в обоих случаях будет равен 0.5, т. е. при вращении кости на 90° вершина изменит положение только на 45. Это только по умолчанию. В идеале вес вы можете настраивать самостоятельно (если позволит используемая программа 3D-графики). Если вершина принадлежит одной кости, то вес ее преобразований равен 1, т. е. она будет трансформироваться на все 100%.

Вот тут отклонюсь от темы и дам один совет — *никогда не двигайте кости для создания анимации*. Все кости только вращаются относительно начала (точки крепления с главной или вышестоящей по иерархии костью). Например, кость руки крепится к плечу, и чтобы протянуть руку вперед, происходит вращение кости относительно плеча. При этом все остальные части руки (локоть, кисть, пальцы) движутся автоматически. Их перемещать не нужно, если не хотите оторвать руку или ногу от туловища.

Попробуйте сейчас подвигать пальцем, и вы наглядно убедитесь, что любые движения пальца — это вращение относительно кисти. Если попытаться дви-

нуть по прямой, то можно выбить палец из сустава. Конечно, бывают индивидуумы с разбитыми суставами, которые производят небольшие прямолинейные движения, но они небольшие (в пределах сустава), и это уже на прием к врачу. Хотя я не врач, и мне кажется это не излечимо, но в любом случае — это не нормально и следует обратиться в больницу.

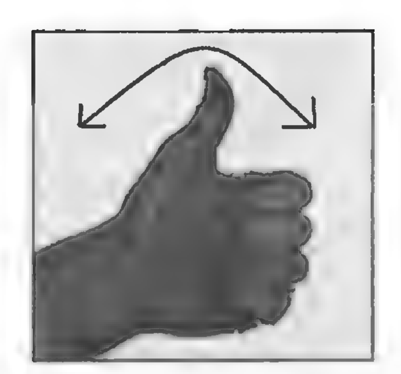


Рис. 3.2. Возможное движение пальца — вращение

Хочу предупредить вас — не пытайтесь с помощью скелета сделать мимику. Тут необходимо движение губ, а в них нет костей, и не пытайтесь их туда совать. Здесь необходим совершенно другой метод, но это уже другая история, которой не стоит забивать сейчас голову.

3.2. Формат хранения сетки

Создав сетку персонажа, скелет и анимацию, все это необходимо сохранить для последующей загрузки из собственной программы. Вот тут возникает проблема выбора — как сохранить? Формат MAX позволяет хранить всю необходимую информацию, но он слишком сложный для использования в собственных программах. Можно использовать старый формат 3DS, но тут возникает проблема с сохранением информации о костях и самой скелетной анимации. Эти проблемы можно обойти, но сделать это достаточно трудно. Придется писать собственные анализаторы и загрузки данных, а потом еще создавать классы или функции для поддержки всего загруженного в программе.

Зачем выдумывать себе проблемы, когда можно использовать формат файла X, который разработан Microsoft и является открытым, универсальным и легко расширяемым. Расширять возможности файла нам не понадобится, потому что все необходимое для хранения сетки и костей в нем уже есть. Сама программа 3ds Max не может сохранять или экспортировать данные в формат X, но можно установить специальный plug-in, разработанный самой Microsoft.

Этот plug-in можно найти в составе DX SDK. Окно этого модуля показано на рис. 3.3.



Рис. 3.3. Plug-in для экспорта модели, скелета и анимации в X-файл

У X-файла есть еще одно преимущество — среди функций Direct3D есть возможность с помощью одной только функции загрузить сетку и еще одной функцией скелет. Корпорация Microsoft здесь очень хорошо позаботилась о нас, как о программистах. Но о функциях чуть позже, давайте еще немного мысленно пощупаем скелет.

Давайте посмотрим, из чего состоит скелет в X-файле и что нам предстоит загружать. Для лучшего понимания я набросал небольшую схему (рис. 3.4). Описывать тело в подробностях я не стал, потому что для понимания материала достаточно и этого. В центре этой схемы находится основная кость, вокруг которой множество других. Основная — это не просто любая или центральная, а именно основная кость. При ее перемещении или вращении вращается абсолютно весь объект. У человека эту роль выполняет позвоночник, а у гуманоида — что захотите ☺. С гуманоидами ничего посоветовать не могу, потому что не знаю, какую игру вы планируете и с какой планеты ваши пришельцы.

Все остальные кости крепятся к основной, создавая иерархическое дерево. Поворачивая любую часть этого скелета, перемещаются все дочерние кости. Такая схема создана только для удобства, чтобы она хоть немного напоминала человека. Если взглянуть на иерархию скелета в памяти, то она будет по-

хожа на дерево с множеством уровней, а не на человека. Если выстроить скелет, т. к. он находится в памяти, то может получиться, что нога будет на одном уровне с грудью (рис. 3.5).

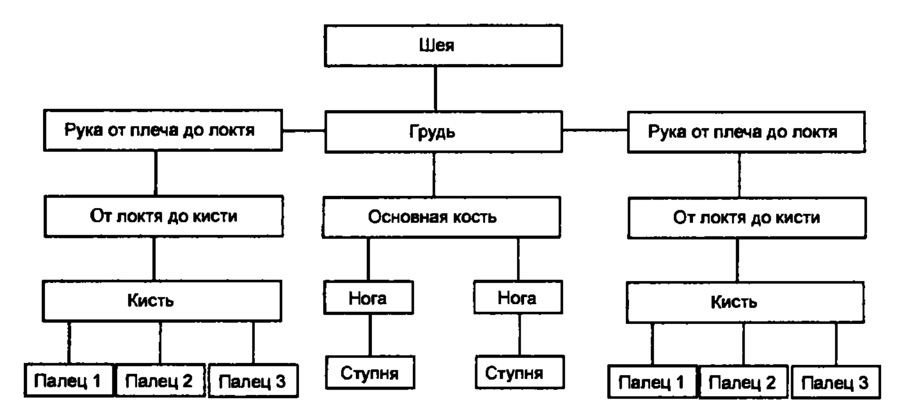


Рис. 3.4. Примерная схема скелета/костей в X-файле

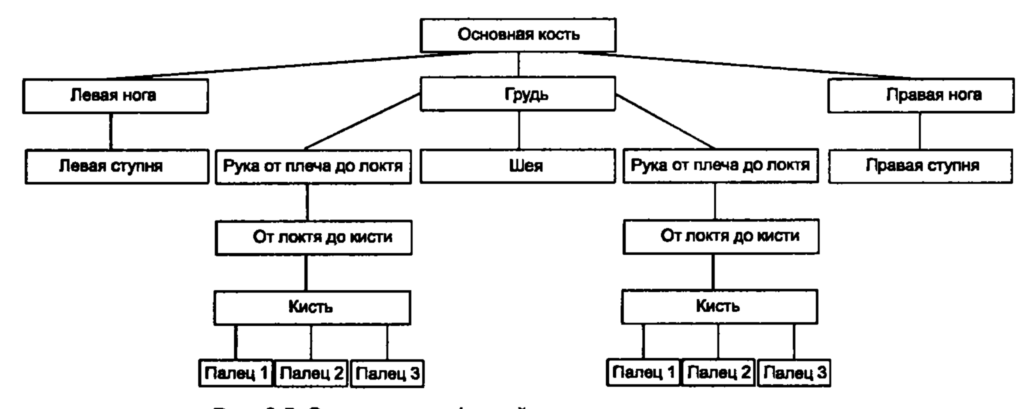


Рис. 3.5. Схема скелета/костей, т. к. она находится в памяти

Может показаться, что так работать с костями будет неудобно, но на самом деле ничего сложного нет. Создавая вторую схему скелета, я всего лишь изменил положение элементов схемы, а связи не изменялись. Все связано в той же последовательности.

Теперь посмотрим непосредственно на сам файл, и как в нем описывается информация о сетке и костях. Да, до этого мы загружали данные из X-файла, но даже не задумывались о том, как они хранятся. А ведь ничего сложного тут нет. Файл формата X может хранить данные как в бинарном, так и в текстовом виде. Бинарный вид лучше тем, что может занимать меньше места на диске, а текстовый на 50% больше. Это за счет большого количества перево-

дов кареток и пустых строк. Но зато текстовый вид можно редактировать вручную в любом текстовом редакторе. Лишь бы этот редактор мог открывать большие файлы, ведь описание 3D-моделей отнимает много места. Например, файл tiny.x занимает 1,5 Мбайта.

Вне зависимости от того, бинарный или текстовый вид у X-файла, данные одни и те же, разница только в представлении, поэтому мы будем рассматривать текстовый вид, как более наглядный. В листинге 3.1 можно увидеть часть файла tiny.x.

Листинг 3.1. Часть содержимого файла tiny.x

```
xof 0303txt 0032
template XSkinMeshHeader {
 <3cf169ce-ff7c-44ab-93c0-f78f62d172e2>
 WORD nMaxSkinWeightsPerVertex;
 WORD nMaxSkinWeightsPerFace;
 WORD nBones;
}

template VertexDuplicationIndices {
 <b8d65549-d7c9-4995-89cf-53a9a8b031e3>
 DWORD nIndices;
 DWORD nOriginalVertices;
 array DWORD indices[nIndices];
}

template SkinWeights {
 <6f0d123b-bad2-4167-a0d0-80224f25fabb>
 STRING transformNodeName;
 DWORD nWeights;
 array DWORD vertexIndices[nWeights];
 array FLOAT weights[nWeights];
 Matrix4x4 matrixOffset;
}


Frame Scene_Root {

 FrameTransformMatrix {
1.000000,0.000000,0.000000,0.000000,0.000000,1.000000,0.000000,0.000000,0
.000000,0.000000,1.000000,0.000000,0.000000,0.000000,0.000000,1.000000;;
 }
```

```
Frame body {


  FrameTransformMatrix {
   1.278853,0.000000,-
0.000000,0.000000,0.000000,0.000000,1.123165,0.000000,0.000000,-
1.470235,0.000000,0.000000,0.135977,2.027985,133.967667,1.000000;;
  }

  Frame {

   FrameTransformMatrix {
    1.000000,-0.000000,-0.000000,0.000000,-
0.000000,1.000000,0.000000,0.000000,-
0.000000,0.000000,1.000000,0.000000,-0.142114,0.000023,-
49.556850,1.000000;;
   }

   Mesh {
    4432;
    -34.720058
    ...
   }
```

Для написания собственного анализатора (мы его будем писать на протяжении этой главы) нам необходимо знать только основы содержимого, потому что DirectX предоставляет нам все необходимые интерфейсы, которые позволят самостоятельно анализировать файл и при этом спрячут всю сложность формата.

Любой файл начинается с заголовка, который должен сообщать анализатору информацию о файле, версию и т. д. В данном случае эту роль выполняет первая строка. Она должна начинаться с букв `xof`, которые подтверждают, что перед нами формат файла X. Иначе это файл другого формата, просто ему дали расширение X. После этого идет версия шаблонов, использованных в файле — `0303`. Если после этого идет `txt`, то данные представлены в текстовом виде, а если `bin`, то в бинарном. И последнее — число определяет точность вещественных чисел. У вещественных чисел может быть переменное число бит после запятой, но в большинстве случаев достаточно иметь точность в 32 бит.

Следующая строка описывает X-файл, описанный шаблонами версией 0303, в бинарном виде и с точностью вещественных чисел в 32 бита:

```
xof 0303bin 0032
```

После этого следует описание шаблонов, которые используются в файле. Описание шаблонов похоже на структуры в языке C++, только вначале стоит ключевое слово template. В листинге 3.1 показано описание трех шаблонов. Давайте рассмотрим первый из них:

```
template XSkinMeshHeader {
 <3cf169ce-ff7c-44ab-93c0-f78f62d172e2>
 WORD nMaxSkinWeightsPerVertex;
 WORD nMaxSkinWeightsPerFace;
 WORD nBones;
}
```

После ключевого слова template идет имя шаблона. В данном случае имя это XSkinMeshHeader. После этого, в фигурных скобках идет описание шаблона. Этот шаблон должен начинаться с уникального идентификатора GUID. После этого следует описание формата. В данном случае описание состоит из трех чисел типа WORD. После типа данных идет описание, по которому вы можете определить, что означает это число.

Запустите поиск имени шаблона по файлу, и вы должны найти следующее использование шаблона XSkinMeshHeader:

```
XSkinMeshHeader {
   2;
   4;
   35;
}
```

Здесь используется шаблон XSkinMeshHeader. Как видите, он состоит из трех целых чисел, разделенных точкой с запятой, как и было в описании.

Мы не будем сейчас работать со специфичными шаблонами и обойдемся только стандартными. Вот некоторые из стандартных шаблонов:

- Mesh (TID_D3DRMMesh) — шаблон содержит данные сетки;
- Vector (TID_D3DRMVector) — шаблон описывает вектор;
- MeshFace (TID_D3DRMMeshFace) — грань сетки;
- Material (TID_D3DRMMaterial) — материал;
- MaterialArray (TID_D3DRMMaterialArray) — массив материалов;
- Frame (TID_D3DRMFrame) — шаблон кадра, описывающего иерархию;
- FrameTransformMatrix (TID_D3DRMFrameTransformMatrix) — матрица положения фрейма;
- MeshTextureCoords (TID_D3DRMMeshTextureCoords) — текстурные координаты сетки Mesh;

- MeshNormals (TID_D3DRMMeshNormals) — нормали сетки;
- Matrix4x4 (TID_D3DRMMatrix4x4) — матрица размером 4×4;
- Animation (TID_D3DRMAnimation) — информация об анимации одного кадра;
- AnimationSet (TID_D3DRMAnimationSet) — набор шаблонов анимации;
- AnimationKey (TID_D3DRMAnimationKey) — информация о ключевом кадре;
- MaterialAmbientColor (TID_D3DRMMaterialAmbientColor) — внешний цвет материала;
- MaterialDiffuseColor (TID_D3DRMMaterialDiffuseColor) — рассеянный цвет материала;
- MaterialSpecularColor (TID_D3DRMMaterialSpecularColor) — отражающий цвет материала;
- MaterialEmissiveColor (TID_D3DRMMaterialEmissiveColor) — цвет излучения;
- ColorRGBA (TID_D3DRMColorRGBA) — цвет в формате RGBA. Есть еще вариант из трех составляющих — ColorRGB;
- TextureFilename (TID_D3DRMTextureFilename) — имя файла текстуры;
- MeshVertexColors (TID_D3DRMMeshVertexColors) — цвета вершин сетки;
- Light (TID_D3DRMLight) — освещение;
- LightRange (TID_D3DRMLightRange) — дальность освещения;
- TID_D3DRMCamera (TID_D3DRMCamera) — камера.

В скобках указаны константы для соответствующих шаблону GUID идентификаторов. Имя этой константы определить достаточно легко. Необходимо просто добавить в начало имени шаблона TID_D3DRM. При создании собственного анализатора X-файлов нам пригодятся эти константы.

Как видите, среди стандартных шаблонов уже есть все необходимое для хранения любых объектов 3D-сцены, к тому же я привел не полный список стандартных шаблонов. И несмотря на это, мы можем расширять возможности этого файла.

3.3. Основы программирования скелетов

Прежде чем писать пример и внедрять поддержку анимированного персонажа в игру, необходимо поговорить о функциях, которые есть в DirectX для поддержки скелетов. Давайте немного времени потратим на теорию, зато потом за пять секунд напишем и рассмотрим практический пример.

Для загрузки информации из X-файла с учетом скелета можно использовать два метода. Первый из них — воспользоваться функцией D3DXLoadMeshHierarchyFromX, которая в общем виде выглядит следующим образом:

```
HRESULT D3DXLoadMeshHierarchyFromX(
 LPCTSTR Filename,
 DWORD MeshOptions,
 LPDIRECT3DDEVICE9 pDevice,
 LPD3DXALLOCATEHIERARCHY pAlloc,
 LPD3DXLOADUSERDATA pUserDataLoader,
 LPD3DXFRAME* ppFrameHeirarchy,
 LPD3DXANIMATIONCONTROLLER* ppAnimController
);
```

Давайте быстренько пробежимся по параметрам этой функции:

- Filename — что-то мне подсказывает, что это путь к загружаемому X-файлу;
- MeshOptions — опции загрузки. Чаще всего требуется опция D3DXMESH_MANAGED, чтобы можно было управлять загруженными данными вершинного буфера. Если управление не нужно, то можно вообще ничего не указывать, чтобы использовать значения по умолчанию;
- pDevice — указатель на интерфейс IDirect3DDevice, я думаю, тут пояснения не нужны, что это такое и зачем;
- pAlloc — указатель на интерфейс ID3DXAllocateHierarchy, методы которого вызываются во время загрузки информации из X-файла. Этот интерфейс необходим для выделения и освобождения фреймов и контейнера объектов;
- pUserDataLoader — указатель на интерфейс ID3DXLoadUserData для загрузки пользовательских данных. Да, X-файл является свободным, гибким, расширяемым и поэтому может содержать и пользовательские данные, которые можно обработать, указав здесь собственный экземпляр интерфейса ID3DXLoadUserData;
- ppFrameHeirarchy — возвращает указатель на иерархию загруженных фреймов в виде массива структур D3DXFRAME;
- ppAnimController — указатель на интерфейс ID3DXAnimationController, в котором будет информация об анимации. Да, в X-файле может быть еще и заранее подготовленная анимация. Нет, ей следовать не обязательно, но иногда очень удобно использовать.

Давайте остановимся подробнее на шестом параметре функции D3DXLoadMeshHierarchyFromX, где мы получаем указатель на массив структур

D3DXFRAME. Что это такое? На самом деле, каждая такая структура описывает определенную кость в скелете и выглядит следующим образом:

```
typedef struct _D3DXFRAME {
    LPTSTR Name;
    D3DXMATRIX TransformationMatrix;
    LPD3DXMESHCONTAINER pMeshContainer;
    struct _D3DXFRAME *pFrameSibling;
    struct _D3DXFRAME *pFrameFirstChild;
} D3DXFRAME, *LPD3DXFRAME;
```

Давайте посмотрим, что есть в этой структуре и для чего.

- `Name` — имя элемента скелета. Каждая кость может иметь свое имя и по нему достаточно удобно находить именно ту часть скелета, которая требует вращения. Зная имя кости, вы легко можете найти ее структуру, перебрав весь скелет. Указатели на наиболее часто используемые структуры лучше сохранить в глобальных переменных, чтобы не перебирать каждый раз весь скелет и повысить скорость анимации.
- `TransformationMatrix` — матрица преобразований, определяющая положение данной косточки.
- `pMeshContainer` — указатель на контейнер сетки.
- `pFrameSibling` — указатель на следующую структуру D3DXFRAME, которая находится на том же уровне. Например, все пальцы находятся на одном и том же уровне иерархии скелета и к ним можно последовательно получить доступ через этот указатель. Если этот параметр равен нулю, то больше костей на этом же уровне нет.
- `pFrameFirstChild` — указатель на первую кость, которая находится на уровень ниже. Если этот параметр равен нулю, то дочерних костей нет.

Структура содержит один большой недостаток, который мы обязаны решить — она не отражает всех необходимых параметров. Например, нет комбинированной матрицы, которая отражала бы реальное положение кости, т. е. комбинацию матриц всех предыдущих костей. Без этой информации нам будет сложно, поэтому расширим структуру, добавив соответствующее поле:

```
struct D3DXFRAME_DERIVED: public D3DXFRAME
{
    D3DXMATRIXA16 CombinedMatrix;
};
```

Теперь в наших примерах будет использоваться именно структура D3DXFRAME_DERIVED.

4 Зак. 543

На первый взгляд все очень легко и прекрасно, но это только на первый взгляд. Посмотрите на функцию D3DXLoadMeshHierarchyFromX и определите, где здесь информация о сетке? Да, информация расположена в не очень удобном формате, поэтому придется что-то с этим делать.

Вместо того чтобы исправлять результат работы D3DXLoadMeshHierarchyFromX, давайте лучше воспользуемся другим методом загрузки сетки с учетом скелета — напишем собственную функцию загрузки, чтобы расположить данные в необходимом виде, тем более что это не так уж и сложно. Хотя нет, функций потребуется несколько, но в любом случае затраты на их создания окупятся, при сопровождении и использовании данного метода.

Формат X открыт, а DirectX предоставляет для нас все необходимые интерфейсы, которые упростят создание анализаторов файла. Я вам предлагаю оптимизированный, достаточно быстрый и очень удобный метод анализа, проверенный мной во времени. В последующих разделах мы рассмотрим этот метод на теории и практике.

3.4. Загрузка сетки и скелета из Х-файла

Создание анализатора строится на "трех китах", а точнее, на трех следующих интерфейсах:

- IDirectXFile — с помощью этого интерфейса вы можете создавать интерфейс IDirectXFileEnumObject и регистрировать шаблоны;
- IDirectXFileEnumObject — с помощью этого интерфейса можно получать данные объекта;
- IDirectXFileData — этот интерфейс используется для построения или прямого доступа к иерархии объектов данных.

Для начала мы должны создать интерфейс IDirectXFile. С помощью его метод RegisterTemplates регистрирует шаблоны Х-файла, которые мы хотим обрабатывать. С помощью этих шаблонов описываются загружаемые данные. После этого запускаем разбор файла с помощью метода CreateEnumObject.

Давайте посмотрим загрузку на примере, и в процессе его разбора познакомимся со всеми тремя интерфейсами и как с ними работать. Таким образом, мы будем одновременно рассматривать теорию и практику, и поэтому сэкономим время и место. Для начала добавим в наш класс управления объектом CDXGObject метод LoadSkinMeshFromFile (листинг 3.2), который будет загружать не только сетку, но и структуру скелета.

Листинг 3.2. Метод для загрузки сетки со скелетом

```
bool CDXGObject::LoadSkinMeshFromFile(char* filename)
{
 SkinMesh = false;
 SkinFrame = NULL;

 // Интерфейсы для загрузки
 IDirectXFile *pxFile;
 IDirectXFileEnumObject *pxEnum;
 IDirectXFileData *pxData;

 // Создать интерфейс DirectXFileCreate
 if (FAILED(DirectXFileCreate(&pxFile)))
  return false;

 // Зарегистрировать стандартные шаблоны
 if (FAILED(pxFile->RegisterTemplates(D3DRM_XTEMPLATES,
      D3DRM_XTEMPLATE_BYTES)))
  return false;

 // Разобрать X-файл
 if (FAILED(pxFile->CreateEnumObject(filename,
      DXFILELOAD_FROMFILE, &pxEnum)))
  return FALSE;

 // Цикл перебора элементов X-файла
 D3DXFRAME_DERIVED **frd = NULL;
 while (SUCCEEDED(pxEnum->GetNextDataObject(&pxData)))
 {
  if (ParseObject(pxData, NULL, frd, FALSE, -1) == FALSE)
   break;
  if (pxData){pxData->Release(); pxData=NULL;}
 }

 // Если сетка загружена, то...
 // Если есть информация о костях
 if (pSkinInfo)
 {
  // Создание массива позиций фреймов и костей
  ppFrameMatrix = new D3DXMATRIX*[pSkinInfo->GetNumBones()];
  pBoneMatrix = new D3DXMATRIX[pSkinInfo->GetNumBones()];
```

```
  for (int i=0; i< pSkinInfo->GetNumBones(); i++)
  {
   const char *BoneName = pSkinInfo->GetBoneName(i);

   D3DXFRAME_DERIVED *pFrame = FindDXFrame(BoneName, SkinFrame);

   if(pFrame)
    ppFrameMatrix[i] = &pFrame->CombinedMatrix;
   else
    ppFrameMatrix[i] = NULL;
  }
 }

 SkinnedMeshPositioning();

 if (pxEnum) {pxEnum->Release(); pxEnum=NULL;}
 if (pxFile) {pxFile->Release(); pxFile=NULL;}
 if (pxData) {pxData->Release(); pxData=NULL;}
 return true;
}
```

Давайте сразу же посмотрим на переменные, которые должны быть добавлены в описание класса объекта в разделе `private`:

```
D3DXMESHDATA MeshData;          // Данные сетки
LPD3DXSKININFO pSkinInfo;       // Информация о скелете

D3DXFRAME_DERIVED *SkinFrame; // Массив фреймов
D3DXMATRIX **ppFrameMatrix;     // Матрицы положения фреймов
D3DXMATRIX *pBoneMatrix;        // Матрицы положения костей
```

Нас всегда учат проверять на ошибки все возможные внештатные ситуации, чтобы программа не вышла из-под контроля. Некоторые настолько ленивы, что не используют проверок вообще. Я считаю, что нужно иметь разумную достаточность, потому в данном примере проверяется далеко не все, но файлы могут отсутствовать (повреждение диска или банальные ошибки при установке) и это следует проконтролировать. Контролировать нужно и существование объектов памяти, поэтому на этих проверках экономить не стоит.

Итак, в самом начале обнуляем переменную `SkinFrame`, которая имеет тип `D3DXFRAME_DERIVED`. После этого объявляются три переменные в виде интерфейсов `IDirectXFile`, `IDirectXFileEnumObject` и `IDirectXFileData`, которые будут использоваться для загрузки:

```
IDirectXFile *pxFile;
IDirectXFileEnumObject *pxEnum;
IDirectXFileData *pxData;
```

Теперь инициализируем интерфейс pxFile. Для этого используется функция DirectXFileCreate, которой необходимо передать переменную типа IDirectXFile, которую в свою очередь нужно проинициализировать.

Теперь регистрируем шаблоны. Для этого используется метод RegisterTemplates интерфейса IDirectXFile. У этого метода имеется два параметра:

- буфер, состоящий из бинарного или текстового формата X-файла, который содержит шаблон;
- размер буфера.

Мы никаких лишних данных загружать не будем и достаточно только стандартного шаблона, который описан в файлах rmxftmpl.h и rmxfguid.h. Подключите оба этих заголовочных файла. В первом можно найти шаблон, а во втором соответствующие константы для идентификаторов GUID, которые мы уже рассматривали в *разд. 3.2*. Стандартный шаблон описан в качестве массива с именем D3DRM_XTEMPLATES, а его размер находится в константе D3DRM_XTEMPLATE_BYTES.

Теперь создаем объект перечислений с помощью метода CreateEnumObject интерфейса IDirectXFile. Этот метод получает следующие параметры:

- pvSource — имя файла, который необходимо открыть и разобрать данные с помощью шаблонов;
- dwLoadOptions — параметры загрузки. Мы указываем значение DXFILELOAD_FROMFILE, т. е. загрузка из файла. Вы можете также указать следующие опции:
  - DXFILELOAD_FROMRESORCE — загружать из ресурса. В этом случае параметр pvSource является структурой DXFILELOADRESOURCE, по которой определяется загружаемый ресурс;
  - DXFILELOAD_FROMMEMORY — использовать файл, загруженный в память. В этом случае параметр pvSource является структурой DXFILELOADMEMORY, по которой определяется адрес памяти, куда загружен файл и размер данных;
  - DXFILELOAD_FROMURL — загрузить данные по указанному адресу URL;
- ppEnumObj — указатель на указатель типа интерфейса IDirectXFileEnumObject, который будет использоваться для перечисления найденных данных файла.

Загрузив данный и создав объект перечисления, мы запускаем цикл, который и будет перебирать все найденные шаблоны и передавать их функции

ParseObject для последующего разбора и при необходимости сохранения для последующего использования. Этот цикл выглядит следующим образом:

```
while (SUCCEEDED(pxEnum->GetNextDataObject(&pxData)))
{
 if (ParseObject(pxData, NULL, frd, FALSE, -1) == FALSE)
  break;
 if (pxData){pxData->Release(); pxData=NULL;}
}
```

Данный цикл выполняется, пока метод GetNextDataObject удачно получает очередной элемент данных X-файла. В качестве параметра этому методу необходимо передать указатель на переменную типа IDirectXFileData. Если данные найдены, то они будут возвращены через этот параметр. Инициализировать ничего не нужно, метод GetNextDataObject сам создаст экземпляр класса, поэтому наоборот, после обработки данных и перед поиском следующего объекта переменную желательно очистить, чтобы не допустить большой утечки памяти.

Получив данные, передаем их функции ParseObject. Эту функцию мы рассмотрим далее в следующем разделе книги, а сейчас нам достаточно знать, что она будет просматривать полученные данные и в зависимости от типа (шаблона) сохранять для последующего использования во время управления и отображения персонажа.

Движемся дальше. После загрузки данных проверяем переменную pSkinInfo. Эту переменную необходимо объявить в нашем классе управления объектами в разделе private следующим образом:

```
LPD3DXSKININFO pSkinInfo;
```

Тип данных LPD3DXSKININFO — это указатель на информацию о скелете, и эту информацию мы будем заполнять во время разбора X-файла. Если информация найдена, то создаем два массива ppFrameMatrix и pBoneMatrix для хранения матриц положения каждой кости. Эти массивы необходимы для оптимизации кода во время отображения сцены, чтобы не пришлось искать каждую кость с помощью рекурсивной функции, которую мы напишем позже. Рекурсия удобна для поиска, но слишком накладна для программы.

После инициализации обеих переменных запускаем цикл, который заполнит массивы значениями. Для поиска очередной кости в иерархии используем рекурсивную функцию FindDXFrame, которую мы напишем позже.

После загрузки всех данных освобождаем выделенную память, потому что в играх и графических программах ее лишней не бывает, и будем хранить каждый бит памяти, как зеницу ока.

3.5. Разбор объектов

В *разд. 3.4* мы загрузили X-файл и запустили перебор всех шаблонов. При этом для разбора найденных данных вызывается функция `ParseObject`, которую мы еще и в глаза-то не видели. Пора сделать это — смотрите листинг 3.3.

Листинг 3.3. Функция `ParseObject` для разбора найденных данных

```
bool CDXGObject::ParseObject(IDirectXFileData *pDataObj,
 IDirectXFileData *pParentDataObj,  D3DXFRAME_DERIVED **frd, bool ref)
{
 const GUID *Type;
 if(pDataObj == NULL)
  return NULL;

 if (FAILED(pDataObj->GetType(&Type)))
  Type = NULL;

 // Тип найденного шаблона - матрица преобразований
 if (*Type == TID_D3DRMFrameTransformMatrix && ref == FALSE && frd)
 {
  if (*frd)
  {
   // Получаем данные, т. е. матрицу преобразований
   if (pDataObj)
   {
    void *TemplateData = NULL;
    DWORD TemplateSize = 0;

    pDataObj->GetData(NULL, &TemplateSize, (PVOID*)&TemplateData);
    (*frd)->TransformationMatrix = *(D3DXMATRIX*)TemplateData;
   }
  }
 }

 // Тип найденного шаблона - фрейм
 if(*Type == TID_D3DRMFrame && ref == FALSE)
 {
  D3DXFRAME_DERIVED *pFrame = new D3DXFRAME_DERIVED();
  pFrame->pFrameSibling = pFrame->pFrameFirstChild = NULL;

  // Получить имя фрейма
  if (pDataObj)
```

```
  {
   pFrame->Name = NULL;
   DWORD Size = 0;

   if (FAILED(pDataObj->GetName(NULL, &Size)))
    return NULL;

   if(Size)
    if ((pFrame->Name = new char[Size]) != NULL)
     pDataObj->GetName(pFrame->Name, &Size);
  }

  // Если переменная frd равна NULL, то создаем объект
  if (frd == NULL)
  {
   pFrame->pFrameSibling = SkinFrame;
   pFrame->pFrameFirstChild = NULL;
   SkinFrame = pFrame;
   frd = &pFrame;
  }
  // Иначе, вставляем фрейм в иерархию
  else
  {
   pFrame->pFrameSibling = (*frd)->pFrameFirstChild;
   (*frd)->pFrameFirstChild = pFrame;
   frd = &pFrame;
  }
 }

 // Найденный шаблон - сетка (mesh)
 if (*Type == TID_D3DRMMesh)
  if (ref == FALSE)
   LoadMeshWithSkin(pDevice, pDataObj, D3DXMESH_SYSTEMMEM);

 return ParseChild(pDataObj, frd, ref);
}
```

Эта функция вызывается тогда, когда найдены новые данные. Эти данные передаются в функцию через первый параметр, где находится указатель на интерфейс `IDirectXFileData`.

Сначала проверяем, равен ли этот параметр нулю. Если да, то разбирать нечего и необходимо прервать работу функции. Если параметр не нулевой, то

данные найдены и через метод GetType определяем тип шаблона. В качестве параметра метода получаем переменную для хранения GUID-идентификатора, и именно по этому GUID мы определяем шаблон.

Мы будем загружать следующие три типа данных:

- фреймы (тип шаблона TID_D3DRMFrame);
- матрицу положения фрейма (TID_D3DRMFrameTransformMatrix);
- данные о сетке (TID_D3DRMMesh).

Наша задача проверить, если это необходимый шаблон, то получить данные с помощью метода GetData, у которого имеется три параметра:

- Имя параметра, данные которого нужно получить. Можно указать NULL, если необходимы все данные.
- Размер буфера для данных.
- Указатель на буфер, через который мы получаем информацию.

Итак, сначала проверяем, равен ли найденный тип матрице. Если да, то получаем ее с помощью следующего кода:

```
void *TemplateData = NULL;
DWORD TemplateSize = 0;

pDataObj->GetData(NULL, &TemplateSize, (PVOID*)&TemplateData);
```

Получив данные, мы сохраняем их в матрицу последнего загруженного фрейма, потому что матрица явно относится к нему:

```
(*frd)->TransformationMatrix = *(D3DXMATRIX*)TemplateData;
```

Да, переменная frd указывает на последний загруженный фрейм и передается в качестве параметра.

Следующим этапом проверяем, если найденный шаблон является фреймом, то нужно создать соответствующую структуру для хранения данных и заполнить ее данными. Сразу после создания фрейма обнуляем указатели на следующий фрейм того же уровня и дочерний фрейм:

```
D3DXFRAME_DERIVED *pFrame = new D3DXFRAME_DERIVED();
pFrame->pFrameSibling = pFrame->pFrameFirstChild = NULL;
```

В шаблоне TID_D3DRMFrame, который был найден, находится имя фрейма, поэтому загружаем его, но в данном случае будем использовать метод GetName, и вызывать его дважды:

```
pFrame->Name = NULL;
DWORD  Size = 0;
```

```
if (FAILED(pDataObj->GetName(NULL, &Size)))
 return NULL;

if(Size)
 if ((pFrame->Name = new char[Size]) != NULL)
  pDataObj->GetName(pFrame->Name, &Size);
```

Методу GetName необходимо передать два параметра — буфер для хранения имени и размер буфера. Но какой размер буфера завести, ведь мы не знаем длину имени? Для этого можно вызвать метод GetName, указав нулевое значение вместо буфера и переменную для хранения размера. В результате, вы получите длину имени. Теперь можно выделять необходимую память и получать непосредственно имя.

После этого проверяем, если еще нет загруженного главного фрейма (переменная frd равна нулю), то текущий ставим на одном уровне с фреймом из переменной SkinFrame, иначе делаем его дочерним.

Если найдена сетка Mesh, то загружаем ее с помощью функции LoadMeshWithSkin, которую мы рассмотрим в следующем *разд. 3.6.*

Прежде чем завершить работу, функция возвращает результат работы ParseChild:

```
return ParseChild(pDataObj, frd, ref);
```

Эта функция разбирает дочерние элементы, и ее мы рассмотрим в *разд. 3.7.*

3.6. Загрузка сетки

При самостоятельном разборе X-файла лучше использовать немного другой метод загрузки сетки, а именно — DirectX функцию D3DXLoadSkinMeshFromXof. Она загружает сетку из интерфейса IDirectXFileData. Но прежде чем рассматривать эту функцию, давайте посмотрим на весь код, который в нашей программе загружает сетку, т. е. на функцию LoadMeshWithSkin (листинг 3.4).

Листинг 3.4. Загрузка сетки

```
HRESULT CDXGObject::LoadMeshWithSkin(IDirect3DDevice9 *pDevice,
   IDirectXFileData *pDataObj, DWORD LoadFlags)
{
 ID3DXMesh *pLoadMesh = NULL;
 pSkinInfo = NULL;
 HRESULT hr;

 ID3DXBuffer *matBuffer = NULL, *abjBuffer = NULL;
```

```
 // Загрузить сетку из открытого файла
 if (FAILED(hr=D3DXLoadSkinMeshFromXof(pDataObj, LoadFlags, pDevice,
  &abjBuffer, &matBuffer, NULL, &dwNumMaterials, &pSkinInfo,
&pLoadMesh)))
  return hr;

 // Удаляем буфер смежности
 if (abjBuffer){abjBuffer->Release();abjBuffer=NULL;}

 // Проверяем наличие скелета и костей
 if (pSkinInfo && ! pSkinInfo>GetNumBones())
  {pSkinInfo->Release(); pSkinInfo=NULL;}

 MeshData.Type = D3DXMESHTYPE_MESH;
 MeshData.pMesh = pLoadMesh; pLoadMesh = NULL;

 if(pSkinInfo)
  MeshData.pMesh->CloneMeshFVF(0, MeshData.pMesh->GetFVF(),
         pDevice, &pObjMesh);

 D3DXMATERIAL *Materials = (D3DXMATERIAL*)matBuffer->GetBufferPointer();
 pMeshMaterials = new D3DMATERIAL9[dwNumMaterials];
 pMeshTextures  = new IDirect3DTexture9*[dwNumMaterials];

 // Загружаем материалы
 for(DWORD i=0;i<dwNumMaterials; i++)
 {
  pMeshMaterials[i] = Materials[i].MatD3D;
  pMeshMaterials[i].Ambient = pMeshMaterials[i].Diffuse;

  // Если есть текстура, то загружаем ее
  if(Materials[i].pTextureFilename)
   D3DXCreateTextureFromFile(pDevice,
        Materials[i].pTextureFilename, &pMeshTextures[i]);
  else
   pMeshTextures[i] = NULL;
 }

 if (matBuffer){matBuffer->Release(); matBuffer=NULL;}

 return S_OK;
}
```

Для загрузки сетки мы используем функцию D3DXLoadSkinMeshFromXof, которая в общем виде выглядит следующим образом:

```
HRESULT D3DXLoadSkinMeshFromXof(
    LPDIRECTXFILEDATA pXofObjMesh,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER* ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD* pMatOut,
    LPD3DXSKININFO* ppSkinInfo,
    LPD3DXMESH* ppMesh
);
```

Параметров получается немало, но самое важное — вы должны быть аккуратны. В DirectX версии 8.1 и 9.0 функции отличаются как небо и земля. Совершенно разные параметры, поэтому при использовании старого DirectX возникнут серьезные проблемы с компиляцией. Лично я не вижу необходимости использовать 8.1, поэтому эту версию функции забудем.

Итак, в качестве параметров функция получает:

- pXofObjMesh — указатель на интерфейс IDirectXFileData, данные которого необходимо загрузить;
- Options — опции загрузки. Мы будем использовать флаг D3DXMESH_SYSTEMMEM для загрузки данных в системную память;
- pDevice — устройство Direct3D;
- ppAdjacency — указатель на интерфейс ID3DXBuffer, для хранения буфера смежности. Этот параметр заполняется массивом по три числа DWORD на каждую грань, которые определяют смежность каждой грани в сетке.
- ppMaterials — указатель на интерфейс ID3DXBuffer, для хранения массива материалов;
- ppEffectInstances — экземпляры эффектов;
- pMatOut — указатель на количество материалов, загруженных в параметре ppMaterials;
- ppSkinInfo — указатель на интерфейс ID3DXSkinInfo, содержащий информацию о скелете;
- ppMesh — указатель на интерфейс ID3DXMesh, т. е. на саму сетку.

После вызова этого метода для нас важными являются следующие данные — сетка, материалы и параметр ppSkinInfo, в котором содержится информация

о скелете. Информацию о скелете мы сохраняем в переменной pSkinInfo, которая объявлена в разделе private класса управления объектом:

```
LPD3DXSKININFO pSkinInfo;
```

Если эта переменная после загрузки сетки равна нулю, то скелета у сетки нет. В качестве контрольного выстрела можно проверить, чтобы количество костей было больше нуля, ведь нам нет смысла содержать объект и скелет, в котором нет костей. Поэтому после загрузки сетки выполняем следующую проверку:

```
if (pSkinInfo && !pSkinInfo->GetNumBones())
  {pSkinInfo->Release();pSkinInfo=NULL;}
```

Тут же удаляем буфер смежности, потому что он нам не нужен. После этого идет разбор полетов, а именно определение материалов и загрузка текстур. Примерно такой же код мы рассматривали в *разд. 1.5*.

3.7. Загрузка дочерних элементов

Теперь подошло время разобрать функцию ParseChild, которая вызывается перед самым завершением работы функции ParseObject. Код этой функции можно увидеть в листинге 3.5.

Листинг 3.5. Разбор дочерних элементов

```
bool CDXGObject::ParseChild(IDirectXFileData *pDataObj,
        D3DXFRAME_DERIVED **frd, bool ref)
{
 IDirectXFileObject *pSubObj = NULL;
 IDirectXFileData  *pSubData = NULL;
 IDirectXFileDataReference *pDataRef = NULL;

 // Цикл, который выполняется пока не переберем все объекты
 while (SUCCEEDED(pDataObj->GetNextObject(&pSubObj)))
 {
  // Определить, чем является объект
  if(SUCCEEDED(pSubObj->QueryInterface(
          IID_IDirectXFileDataReference,(void**)&pDataRef)))
  {
   // Объект является ссылкой, поэтому преобразовываем
   // его и вызываем метод ParseObject для разбора
   if(SUCCEEDED(pDataRef->Resolve(&pSubData)))
   {
```

```
    if (!ParseObject(pSubData, pDataObj, frd, TRUE))
     return FALSE;
    if (pSubData){pSubData->Release();pSubData=NULL;}
   }
   if (pDataRef){pDataRef->Release();pDataRef=NULL;}
  }
  else
   // Объект является данными, получить их и разобрать
   if (SUCCEEDED(pSubObj->QueryInterface(IID_IDirectXFileData,
         (void**)&pSubData)))
   {
    if (!ParseObject(pSubData, pDataObj, frd, ref))
     return FALSE;

    if (pSubData){pSubData->Release();pSubData=NULL;}
   }

    if (pSubData){pSubData->Release();pSubData=NULL;}
 }

 return TRUE;
}
```

Логика работы функции проста. Сначала определяем следующий дочерний объект с помощью метода GetNextObject. Получив данные, необходимо проверить, являются ли они ссылкой (reference) или бинарным объектом (binary object), с помощью следующей строки кода:

```
if (SUCCEEDED(pSubObj->QueryInterface(
    IID_IDirectXFileDataReference,(void**)&pDataRef)))
```

Если перед нами ссылка, то ее необходимо преобразовать с помощью метода Resolve следующим образом:

```
if (SUCCEEDED(pDataRef->Resolve(&pSubData)))
```

Если результат удачен, то вызываем функцию ParseObject, которую мы уже рассмотрели в *разд. 3.5*. Если перед нами данные, то получаем их, и снова запускаем функцию ParseObject для разбора этих данных.

3.8. Поиск фрейма

Чтобы изменить положение кости, нам необходимо иметь удобный способ поиска фрейма. *Фреймы* — это структуры D3DXFRAME_DERIVED, которые выстроены в иерархии. Как и по каким параметрам мы можем искать фрейм? Ну

конечно же по его имени. Например, положение плечевой кости левой руки определяет фрейм с именем Bip01_L_UpperArm.

В листинге 3.6 показан код функции `FindDXFrame`, которая рекурсивно перебирает все фреймы иерархии в поисках указанного имени. Функция получает в качестве параметров имя искомого фрейма и указатель на фрейм, с которого нужно начать перебирать иерархию.

Листинг 3.6. Поиск кости

```
D3DXFRAME_DERIVED * CDXGObject::FindDXFrame(const char *FrameName,
    D3DXFRAME_DERIVED *Frame)
{
 D3DXFRAME_DERIVED *tempFrame;

 // Если имя совпадает с текущим, то возвращаем его
 if(Frame && Frame->Name && FrameName)
  if (!strcmp(FrameName, Frame->Name))
   return Frame;

 // Если есть фреймы того же уровня, то перебираем их
 if (Frame->pFrameSibling)
  if ((tempFrame = FindDXFrame(FrameName,
                 (D3DXFRAME_DERIVED*)Frame->pFrameSibling)))
   return tempFrame;

 // Если есть дочерние фреймы, то перебираем их
 if (Frame->pFrameFirstChild)
  if((tempFrame = FindDXFrame(FrameName,
                 (D3DXFRAME_DERIVED*)Frame->pFrameFirstChild)))
  return tempFrame;

 return NULL;
}
```

Вначале проверяем, равно ли имя текущего фрейма искомому имени. Если да, то возвращаем его. Если нет, то сначала проверяем, есть ли связанные кости текущего уровня. Если да, то вызываем функцию `FindDXFrame`, для поиска кости в ее иерархии. Если и здесь ничего не найдено, то проверяем, есть ли необходимый фрейм среди дочерних к текущему.

Таким образом, мы рекурсивно перебираем все фреймы. Но рекурсия слишком накладна для процессора, потому что это цикл, который множество раз вызывает одну и ту же функцию. Во время каждого вызова программе прихо-

дится "поднимать" параметры и адрес возврата в стек, производить переход по указанному адресу, а по завершению поиска в обратном порядке производится возврат из функций. Если иерархия длинная, то скорость поиска может быть слишком низкой.

3.9. Обновление сетки

Давайте теперь посмотрим алгоритм, который позволит двигать скелет. Во время движения чаще всего необходимо изменять матрицу преобразования не только одной косточки (например, предплечья), но и всех дочерних (всех костей, которые входят в руку). В этом случае приходится создавать функцию, которая должна выполнять два действия: для начала необходимо изменять положение текущей кости, а после этого вызывается рекурсивная функция. Внутри рекурсивной функции нужно изменять матрицы положения не только дочерних, но и родственных (находящихся на одном уровне) костей. Например, двигая кисть, должны двигаться и все пять пальцев, которые находятся на одном уровне, а не один, первый попавшийся палец. Получается, что преобразование должно происходить примерно так же, как и поиск. Функция обновления матрицы преобразований может выглядеть, как показано в листинге 3.7.

Листинг 3.7. Установка матрицы преобразований

```
void CDXGObject::SetMatrix(
    D3DXMATRIX *matTransformation, D3DXFRAME_DERIVED *Frame)
{
 // Комбинируем текущую матрицу с переданной в параметре
 Frame->CombinedMatrix =
    Frame->TransformationMatrix * (*matTransformation);

 // Изменяем матрицу всех костей этого же уровня
 if (Frame->pFrameSibling)
  SetMatrix(matTransformation,
          (D3DXFRAME_DERIVED*)Frame->pFrameSibling);

 // Изменяем матрицу всех дочерних костей
 if (Frame->pFrameFirstChild)
  SetMatrix(&Frame->CombinedMatrix,
   (D3DXFRAME_DERIVED*)Frame->pFrameFirstChild);
}
```

Обратите внимание, что во время вызова функции обновления родственных костей в качестве второго параметра передается не измененная матрица, а при обновлении дочерних костей уже трансформированный вариант.

Работая с матрицами, вы должны учитывать, что, модифицировав матрицу преобразований одной кости, могут возникнуть проблемы с получением первоначального состояния. Дело в том, что в любом случае есть какая-то погрешность, которая со временем начинает влиять на качество сцены. Например, ноги у человека уже больше не будут выравниваться в тот момент, когда вы хотите остановить свой персонаж и поставить его ровно. Избежать такого эффекта легко. Можно расширить структуру D3DXFRAME_DERIVED, добавив еще одно поле типа D3DXMATRIX. В этом поле можно хранить матрицу преобразований кости на этапе загрузки и при каждом отображении сцены "плясать" именно от этого значения.

Для повышения производительности можно заранее рассчитать ключевые кадры, по которым будет строиться дальнейшая анимация. Таким образом, можно заранее подготовить траектории движения основных телодвижений — ходьба, бег, приседание, прыжок и т. д. Во время игры в зависимости от ситуации останется только воспользоваться уже рассчитанными траекториями, которые не занимают много места и могут использоваться для различных персонажей одновременно. Но не забывайте, что походки у людей отличаются, особенно у мужчин и женщин, но даже если вы на это закроете глаза, то в динамичной сцене игрок не заметит подвоха.

3.10. Обновление сетки

Разговор о скелетах уже достаточно затянулся и создается впечатление, что скелеты достаточно сложная тема. Тема не сложная, просто она достаточно большая. Из-за отсутствия хорошей функции загрузки фреймов нам пришлось потратить немного времени (у меня написание анализатора заняло несколько часов и два дня оптимизации ☺), но зато теперь мы сможем его использовать в любых будущих проектах.

В принципе, наш пример уже готов и может отобразить фигуру персонажа. Достаточно только заменить функцию загрузки персонажа во время инициализации (в функции CGraphEngine::InitObject) с LoadMeshFromFile на LoadSkinMeshFromFile.

Но сетка пока не связана со скелетом. Остался последний шаг — мы должны как-то изменить один из фреймов, чтобы повернуть какую-то часть тела и модифицировать сетку с учетом изменений в скелете. Для этого создадим в классе управления объектами игры функцию SkinnedMeshPositioning, которая будет поворачивать левую руку:

```
void CDXGObject::SkinnedMeshPositioning()
{
 D3DXMATRIX Rot, Rot1;
```

```
 // Повернуть сцену
 D3DXMatrixRotationX(&Rot, 1.5f);

 // Повернуть руку
 D3DXFRAME_DERIVED *r = FindDXFrame("Bip01_L_UpperArm", SkinFrame);
 D3DXMatrixRotationX(&r->TransformationMatrix, -1.5f);

 // Обновить сетку
 SetMatrix(&Rot, SkinFrame);
 UpdateMesh();
}
```

Эта функция не универсальна, и введена только для иллюстрации работы со скелетной сеткой. Если отобразить персонаж tiny.x сразу после загрузки, то он окажется подвешенным в воздухе параллельно земле. Чтобы исправить это, необходимо повернуть весь объект примерно на 90°. Для этого объявляется матрица `Rot`, которая поворачивается на 1.5 по оси X.

Теперь повернем левую руку персонажа на 90°, только в противоположную сторону, чтобы она поднялась вверх. Для этого сначала определяем фрейм левой руки, для чего запускаем поиск по имени Bip01_L_UpperArm. Затем поворачиваем матрицу преобразований найденного фрейма на –1,5.

Матрицы подготовлены, и их необходимо обновить. Для этого вызываем функцию `SetMatrix`. В качестве параметра указываем первый фрейм в иерархии, чтобы функция обновила всю иерархию.

Матрицы готовы, и теперь они должны как-то повлиять на сетку. Для этого вызываем метод `UpdateMesh`, код которого можно увидеть в листинге 3.8.

Листинг 3.8. Обновление сетки

```
HRESULT CDXGObject::UpdateMesh()
{
 void *SrcPtr, *DestPtr;

 for(DWORD i=0;i<pSkinInfo->GetNumBones();i++)
 {
  pBoneMatrix[i] = (*pSkinInfo->GetBoneOffsetMatrix(i));

  pBoneMatrix[i] *= (*ppFrameMatrix[i]);
 }

 MeshData.pMesh->LockVertexBuffer(D3DLOCK_READONLY, (void**)&SrcPtr);
 pObjMesh->LockVertexBuffer(0, (void**)&DestPtr);
```

```
  pSkinInfo->UpdateSkinnedMesh(pBoneMatrix, NULL, SrcPtr, DestPtr);

  pObjMesh->UnlockVertexBuffer();
  MeshData.pMesh->UnlockVertexBuffer();

  return S_OK;
}
```

В этой функции запускается цикл, который перебирает все кости. Внутри цикла получаем матрицу преобразований кости и перемножаем ее с матрицей фрейма.

После этого блокируем сетку и обновляем ее с помощью метода `UpdateSkinnedMesh`. В общем виде этот метод выглядит следующим образом:

```
HRESULT UpdateSkinnedMesh(
  CONST D3DXMATRIX *pBoneTransforms,
  CONST D3DXMATRIX *pBoneInvTransposeTransforms,
  PVOID pVerticesSrc,
  PVOID pVerticesDst
);
```

Здесь у нас имеется четыре параметра:

- `pBoneTransforms` — массив матриц преобразований костей;
- `pBoneInvTransposeTransforms` — массив транспозиций преобразований костей;
- `pVerticesSrc` — буфер, из которого нужно брать вершины;
- `pVerticesDst` — буфер, в который будут помещены преобразованные данные.

Вот и все. На этом рассмотрение примера можно считать оконченным. Результат работы программы показан на рис. 3.6. Мы не двигали никаких точек, зато получили объект человека, рука которого поднята вверх. Изменяя угол одной только матрицы преобразований, изменяется положение множества связанных точек. Таким образом, мы получаем очень хороший способ управления персонажем. Необходимо только добавить в наш движок объекта возможности анимации персонажа, но это уже совсем другой разговор.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter3\Skin.



Рис. 3.6. Результат поднятия руки

3.11. Анимация скелета

Создать просто скелет мало, для игры нам необходимо научить его плавно двигаться, потому что статичные персонажи — это слишком большая редкость. Давайте поговорим об анимации персонажа с помощью анимации скелета.

Современные игры максимально используют скелетную анимацию. Но это не значит, что вы должны математически во время выполнения игры просчитывать шаги на основе законов анатомии, а во время прыжков учитывать земное притяжение. Такие расчеты отнимут слишком много времени и не имеют смысла.

Намного проще и эффективнее заранее просчитать матрицы для контрольных точек и потом через эти точки анимировать скелет. Такую анимацию называют основанной на ключевых кадрах и ее используют не только в играх, но и в графических пакетах, таких как 3D Studio Max или Macromedia Flash. Ес-

ли вы работали с этими пакетами и создавали в них анимацию, то у вас вообще не возникнет никаких проблем с пониманием этого раздела.

Что такое анимация на основе ключевых кадров? Для начала, вспомним, как зародилась эта технология. Все начиналось с мультипликации, когда каждый кадр рисовался в отдельности. С появлением компьютера появилась возможность программно создавать переходы между кадрами. Например, вам необходимо сделать так, чтобы рисованный самолет передвинулся с точки 1 в точку 10 по оси X за время равное 5 с. Точка 1 и точка 10 являются ключевыми кадрами, которые определяют начальное и конечное положение. Движение между этими кадрами мы должны и можем анимировать программно.

Я не художник, но на рис. 3.7 попытался изобразить движение руки человека. Пунктирной линией показаны положения руки, в которых должен быть ключевой кадр.

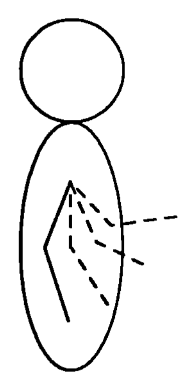
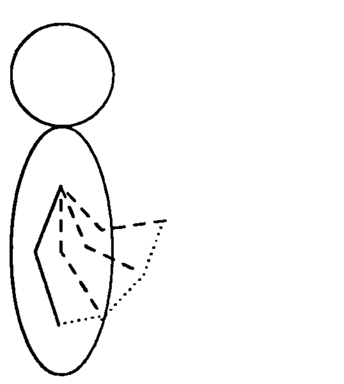


Рис. 3.7. Ключевые кадры движения человека



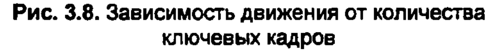


Рис. 3.8. Зависимость движения от количества ключевых кадров

А зачем так много кадров? Неужели нельзя обойтись двумя — начальной и конечной точкой. Ответ прост, мы просто не сможем воссоздать плавную траекторию. Посмотрите на рис. 3.8. Слева линией из точек показано движение руки из четырех ключевых кадров, а справа движение из двух кадров. Обратите внимание, что слева движение ближе к дуге, а справа чисто прямая линия.

Чем больше ключевых кадров, тем более плавное движение мы можем создать. С другой стороны, каждый лишний кадр — это расходы памяти. Да, матрица преобразований занимает не очень много места, поэтому сильно экономить не нужно, но и слишком много матриц тоже создавать не следует. Посмотрите на рис. 3.9, где показано движение по кругу. Траекторию я изобразил пунктирными линиями, а контрольные точки, через которые происходит движение, показаны на круге в виде засечек.

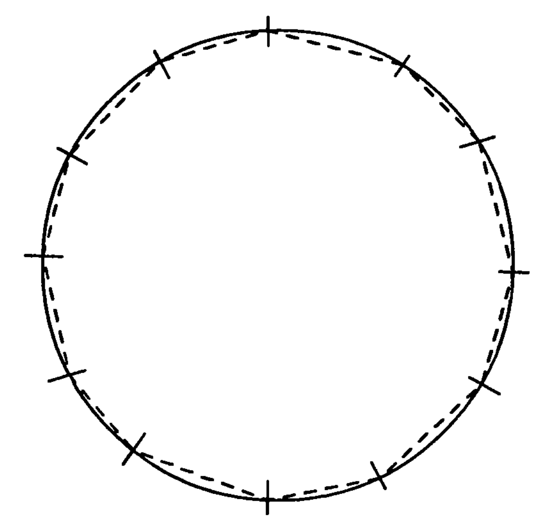


Рис. 3.9. Траектория движения по кругу

При использовании двенадцати контрольных точек траектория проходит достаточно близко к линии круга. Если увеличить число точек в два раза, то траектория будет еще ближе. Дальнейшее увеличение мне кажется уже бессмысленным и в целях экономии можно обойтись и 12-ю ключами.

Положение объекта в ключевые моменты времени определяется ключевыми матрицами. Но что делать в остальные моменты времени? Нельзя же анимировать объект скачками. В книге [4] я рассматривал метод синхронизации во времени. Этот же метод можно использовать и при анимации. Давайте создадим новый проект и рассмотрим пример анимации движения руки персонажа на 90°. Откройте проект из *разд. 3.10*, сейчас мы его расширим.

Для начала в заголовочном файле движка игры объявим две закрытые (`private`) переменные для хранения начала времени анимации, матриц преобразования:

```
DWORD tStartAnimTime;      // Время начала анимации
D3DXMATRIX matAnimStart; // Ключ 1 - начало анимации
           matAnimEnd;   // Ключ 2 - конец анимации
```

Как видите, у нас в этом примере только две матрицы для хранения начального и конечного положения, т. е. мы ограничимся только двумя ключами.

Теперь переходим в методу `InitObject`, где происходит инициализация объектов игры. Модифицируем этот метод следующим образом:

```
bool CGraphEngine::InitObject(int type, int index)
{
  ...
```

```
  case SO_DX_MAN:
    // Здесь идет загрузка сетки и эффекта персонажа
   ...
   D3DXMatrixIdentity(&matAnimStart);
   D3DXMatrixRotationX(&matAnimEnd, -1.5f);
   tStartAnimTime = GetTickCount();
   break;
  ...
}
```

В блоке `case` после загрузки персонажа и установки эффекта устанавливаем следующие три параметра анимации:

- в качестве матрицы начального положения используем единичную матрицу, чтобы рука была вдоль туловища;
- в качестве матрицы конечного положения повернем руку на –1.5. Не забываем, что матрицы костей мы только поворачиваем и никогда не двигаем;
- начало анимации будет текущее время.

В реальных условиях время анимации и матрицы ключевых положений желательно хранить в каком-либо файле и загружать в буфер.

Теперь сама анимация. В самом начале функции `RenderScene` добавляем код из листинга 3.9, для расчета положения руки с учетом прошедшего времени анимации.

Листинг 3.9. Расчет матрицы положения кости с учетом времени анимации

```
void CGraphEngine::RenderScene()
{
 // Текущее время
 double tCurrentTime = GetTickCount();

 // Найти кость руки, которую нужно анимировать
 D3DXFRAME_DERIVED *r =
   objects[1]->FindDXFrame("Bip01_L_UpperArm", objects[1]->SkinFrame);

 // Если прошло время анимации, то начнем заново
 if ((tCurrentTime-tStartAnimTime)>4000)
  tStartAnimTime = GetTickCount();

 // Расчет матрицы положения кости с учетом прошедшего времени
 // анимации и на основе ключевых кадров
 D3DXMATRIX matDiff = matAnimEnd-matAnimStart;
```

```
    matDiff*=(tCurrentTime - tStartAnimTime) / 4000;
    matDiff+=matAnimStart;
    r->TransformationMatrix = matDiff;

    // Матрица положения всего объекта для обновления сетки
    D3DXMATRIX Rot;
    D3DXMatrixRotationX(&Rot, 1.5f);

    // Устанавливаем матрицу и обновляем сетку
    objects[1]->SetMatrix(&Rot, objects[1]->SkinFrame);
    objects[1]->UpdateMesh();

    // Отображение объектов игры
    ...
}
```

Давайте кратко посмотрим, что здесь происходит. Сначала определяем текущее время и ищем фрейм руки, который необходимо повернуть. После этого проверяем, если с начала анимации прошло больше 4 с (4000 мс), то начать анимацию заново. Да, именно 4 с должна происходить данная анимация и это значение вместе с матрицами должно быть загружено из какого-то хранилища.

После этого определяем разницу между матрицей конечного и начального положения. Теперь эту матрицу умножаем на количество времени, прошедшее с начала старта анимации (`tCurrentTime – tStartAnimTime`), деленное на время анимации и прибавляем матрицу начала движения.

Зачем сначала находить разницу между матрицами начального и конечного ключа, а в самом конце опять прибавлять начальную матрицу? Об этом я говорил в книге [4], когда рассматривал синхронизацию демонстрационных роликов во времени на подобном алгоритме. Проблема связана с результатом произведения и сложения матриц. Тут нужно быть очень аккуратным.

После получения необходимого преобразования устанавливаем его фрейму, устанавливаем матрицу преобразования для персонажа в целом, поворачивая его примерно на 90°, и обновляем скелет. Как видите, реализовать скелетную анимацию очень просто и именно сейчас проявляются все преимущества скелета и затраченного времени на изучение этой главы. Запустите пример и убедитесь, что он работает корректно и персонаж машет вам рукой.

Формат файла X позволяет хранить и заранее просчитанную анимацию. Помните, в *разд. 3.2* мы рассматривали шаблоны, среди которых были `TID_D3DRMAnimation` и `AnimationKey`. Но мне не очень понравилась возможность совмещения анимации, скелета и сетки в одном файле. В большом про-

екте слишком неудобно становится управление, поэтому лучше разработать собственный способ хранения данных. Это не так уж и сложно, необходимо просто сохранять в файле в определенном виде матрицы преобразований, которые будут символизировать ключевые кадры движения. Помимо матриц нужно еще время, за которое должно произойти преобразование от одного ключа к другому.

Я не стал рассматривать здесь никаких вариантов хранения данных об анимации, чтобы у вас была возможность поэкспериментировать самостоятельно на досуге. Это позволит лучше понять тему. Попробуйте еще заставить персонажа ходить и прыгать. При этом лучше учитывать хотя бы простые законы физики, чтобы сцена выглядела более реалистично, а именно, не забывайте о силе притяжения. Если о сопротивлении воздуха можно и забыть, потому что эта сила достаточно маленькая и при коротких прыжках человека оказывает незначительное влияние, то прыжок лучше рассчитывать на основе силы толчка и силы притяжения.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter3\AnimationSkin.

3.12. Анимация из X-файла

Несмотря на то, что я люблю использовать собственный формат хранения информации об анимации, мы все же рассмотрим стандартный способ, предложенный разработчиками Microsoft. Этот способ обладает следующими двумя преимуществами, с которыми трудно поспорить:

- анимация и сам объект находятся в одном файле. Вам не нужно придумывать собственный формат хранения данных, все уже сделано за вас, причем очень хорошо сделано;
- для создания анимационных ключей и сохранения их в файл можно использовать 3D-редактор. Таким образом, не придется писать собственную программу сохранения информации, а визуальные возможности 3D-редактора упростят работу аниматора.

Если в вашей команде разработчиков игр есть профессиональный аниматор, то использование анимации, которую можно сохранить в X-файле, для вас окажется очень удобной возможностью, поэтому я не могу опустить эту тему.

Итак, давайте сначала посмотрим, как хранится информация об анимации в X-файле. Для этого возьмем уже затасканный нами файл tiny.x. В листинге 3.10 вы можете видеть пример части файла tiny.x, описывающий анимацию.

Листинг 3.10. Пример информации из X-файла об анимации

```
AnimationSet {
 Animation {
  AnimationKey {
   4;
   2;
   0;16;1.000000,0.000000,0.000000,0.000000,0.000000,1.000000,
   0.000000,0.000000,0.000000,0.000000,1.000000,0.000000,0.000000,
   0.000000,0.000000,1.000000;;,

   4960;16;1.000000,0.000000,0.000000,0.000000,0.000000,1.000000,
   0.000000,0.000000,0.000000,0.000000,1.000000,0.000000,0.000000,
   0.000000,0.000000,1.000000;;;
  }
  { Scene_Root }
 }

 Animation {
  AnimationKey {
  }
 }
}
```

Во главе всего стоит блок `AnomationSet`. Это набор анимации, который описывает определенное действие, например, ходьбу, бег, прыжок, приседание и т. д. В файле tiry.x есть только один набор анимации.

Внутри `AnomationSet` может быть один или несколько блоков `Animation`, которые описывают анимацию и содержат ключи анимации `AnimationKey`. Ключи — это самое сложное, с чем нам предстоит разобраться, и они описывают преобразования определенных костей. Давайте по частям разберем, из чего состоит этот ключ:

```
AnimationKey {
 4;  // Тип ключа
 1;  // Количество ключей
 0;  // Время
 16; // Количество данных
  1.000000,0.000000,0.000000,0.000000,0.000000,1.000000,
  0.000000,0.000000,0.000000,0.000000,1.000000,0.000000,
  0.000000,0.000000,0.000000,1.000000;;,
 { Scene_Root }
}
```

Чтобы удобней было рассматривать структуру ключа, я разбил основные его части на отдельные строки, хотя внутри файла данные будут выглядеть немного по-другому.

Первое число в блоке `AnimationKey` определяет тип ключа. Существует всего четыре типа:

- 0 — такой ключ определяет вращение;
- 1 — ключ содержит данные масштабирования;
- 2 — ключ определяет перемещение;
- 4 — шестнадцать значений, определяющие матрицу преобразований.

Обратите внимание, что индекса 3 не существует. Мы будем использовать только последний ключ. Да, в tiny.x вся анимация строится только матрицами преобразований, и я вам советую тоже ограничиваться только ими. Дело в том, что с помощью матрицы вы можете сделать любое преобразование, а значит, это необходимые и достаточные данные. Если вы будете использовать все типы ключей, то код загрузки и поддержки анимации усложнится, а производительность программы снизится.

После типа ключа идет число, которое определяет количество ключей в данном блоке `AnimationKey`. В данном случае будет только один ключ типа 4, т. е. одна матрица.

Следующее число определяет время ключа. В определенный момент времени указанная кость должна будет находиться в означенной позиции. Очередное число определяет количество данных. В этом случае у нас используется тип данных 4 (матрица), а значит, необходимо 16 значений.

Первые четыре числа имеют определенный смысл и являются целыми числами. После этого идут данные матрицы преобразований или вектора (если ключ имеет тип 0, 1 или 2), которые имеют тип `float`.

3.13. Загрузка анимации из X-файла

Теперь давайте посмотрим, как можно загрузить данные. Для этого расширим пример, который мы закончили писать в *разд. 3.11*. Для начала давайте подготовим переменные, в которые будет происходить загрузка. В предыдущем разделе мы увидели, что данные об анимации хранятся в виде дерева из трех уровней:

```
AnimationSet - Animation - AnimationKey
```

Для удобства, в нашей программе каждому уровню будет сопоставлен определенный класс.

3.13.1. Необходимые классы

Первый класс будет CAnimationKey, задача которого будет хранить время и матрицу преобразования:

```
class CAnimationKey
{
public:
    DWORD Time;           // Время
    D3DXMATRIX m_matKey;  // Матрица преобразований
};
```

Следующий класс CAnimation будет хранить информацию из блока Animation в X-файле. Какую именно информацию, смотрите по комментариям в следующем объявлении:

```
class CAnimation
{
public:
 char *sName;                     // Имя анимации
 D3DXFRAME_DERIVED  *Bone;        // Анимируемая кость
 int numKeys;                     // Количество ключей
 CAnimationKey *animationKey;     // Ключи анимации

 CAnimation  *NextAnim;           // Указатель на следующую анимацию

 CAnimation();                    // Конструктор
 ~CAnimation();                   // Деструктор
};
```

Приведу некоторые пояснения. Когда мы рассматривали файл, то вы видели, что в один блок Animation может входить несколько ключей. В переменной numKeys мы сохраняем количество ключей, а переменная animationKey будет указывать на последовательно выстроенные в памяти ключи.

Переменная NextAnim указывает на следующую найденную анимацию. Если она равна нулю, то следующей анимации нет. Таким образом, мы создаем динамический массив для хранения всех блоков анимации Animation.

И последний класс, который нам понадобится для хранения информации об анимации — CAnimationSet:

```
class CAnimationSet
{
public:
 char *sName;                     // Имя набора анимации
 DWORD Time;                      // Время анимации
```

```
 CAnimationSet *NextAnimSet;  // Следующий набор анимации

 int NumAnimations;           // Количество анимации
 CAnimation  *Animations;     // Анимации

 CAnimationSet();             // Конструктор
 ~CAnimationSet();            // Деструктор
};
```

Тут требуется некоторое пояснение. Дело в том, что у блока `AnimationSet` нет времени анимации. Откуда же мы его можем получить? Все очень просто — в процессе загрузки ключей мы будем проверять их время и в поисках максимального значения. Именно это и будет конечным значением.

Реализация классов проста, потому что у нас есть только конструкторы и деструкторы. В конструкторах переменные соответствующего класса обнуляются, а в деструкторе уничтожаются.

3.13.2. Анализ файла

Все готово, и можно переходить к корректированию функции `ParseObject`. В ней у нас предусмотрена загрузка скелета и сетки, но не анимации. Сейчас исправим эту ситуацию.

Для начала добавим обработку блока `AnimationSet`, который соответствует типу `TID_D3DRMAnimationSet`. Для этого в конец метода `ParseObject` добавляем следующий код:

```
if (*Type == TID_D3DRMAnimationSet)
{
 // Создание и заполнение основных полей объекта CAnimationSet
 CAnimationSet *AnimSet = new CAnimationSet();
 AnimSet->NextAnimSet = AnimationSets;
 AnimationSets = AnimSet;
 NumAnimationSets++;

 // Попытка определить имя набора анимации
 DWORD Size = 0;
 pDataObj->GetName(NULL, &Size);

 if (Size)
 {
  if ((AnimSet->sName = new char[Size]) != NULL)
   pDataObj->GetName(AnimSet->sName, &Size);
 }
```

```
 else
  AnimSet->sName = "Default";
}
```

Если найден блок AnimationSet, то создаем объект CAnimationSet. Сразу же в поле NextAnimSet сохраняем указатель на последний загруженный блок AnimationSet. Если еще ничего не загружалось, то значение этой переменной будет равно нулю. После этого, в переменной AnimationSet сохраняем только что созданный объект CAnimationSet и увеличиваем значение счетчика NumAnimationSets на 1.

После этого пытаемся определить имя текущей анимации. Если попытка заканчивается неудачей, то в качестве имени используем Default. На этом загрузку набора можно считать завершенной.

3.13.3. Анализ блока анимации

Следующий блок, который мы должны загрузить, — Animation, которому соответствует константа TID_D3DRMAnimation. Тут все еще проще, потому что нужно только создать объект и поместить его в массив:

```
if (*Type == TID_D3DRMAnimation)
{
 CAnimation *Anim = new CAnimation();
 Anim->NextAnim = AnimationSets->Animations;
 AnimationSets->Animations = Anim;
 AnimationSets->NumAnimations++;
}
```

Но у анимации может быть еще и имя! Точнее сказать, оно обязано быть, ведь по нему определяется, какая именно кость анимируется данным блоком Animation. Имя определяется в совершенно другом блоке TID_D3DRMFrame. Точно такой же блок мы уже рассматривали в *разд. 3.5*, но тогда он использовался для загрузки информации о скелете, а сейчас для определения имени анимируемой кости. Как же определить разницу между двумя этими случаями? Очень просто — необходимо определить тип родительского объекта. Если это TID_D3DRMAnimation, то загружаем имя анимируемой кости.

Все вышесказанное показано в листинге 3.11, который также должен быть добавлен в метод ParseObject.

Листинг 3.11. Определение имени анимируемой кости

```
if (*Type == TID_D3DRMFrame)
{
 const GUID *parentType = NULL;
```

```
 if (pParentDataObj)
 {
  pParentDataObj->GetType(&parentType);
  if (pParentDataObj && *parentType ==  TID_D3DRMAnimation)
  {
   DWORD Size = 0;
   pDataObj->GetName(NULL, &Size);
   if(Size)
   {
    if ((AnimationSets->Animations->sName = new char[Size]) != NULL)
     pDataObj->GetName(AnimationSets->Animations->sName, &Size);
   }
   else
    AnimationSets->Animations->sName = "Default";
  }
 }
 return TRUE;
}
```

3.13.4. Анализ ключа

Последнее, что нам необходимо сделать, — это определить данные ключа, т. е. время и матрицу преобразований ключа. Эти данные находятся в блоке `AnimationKey` X-файла, что соответствует константе `TID_D3DRMAnimationKey`.

Загрузку ключа можно увидеть в листинге 3.12.

Листинг 3.12. Анализ блока `AnimationKey`

```
if (*Type == TID_D3DRMAnimationKey)
{
 CAnimation *Anim = AnimationSets->Animations;
 DWORD *DataPtr = NULL;
 DWORD TemplateSize = 0;
 pDataObj->GetData(NULL, &TemplateSize, (PVOID*)&DataPtr);

 if (*DataPtr==4)
 {
  DataPtr++;
  Anim->numKeys = *DataPtr;
  DataPtr++;
  Anim->animationKeys = new CAnimationKey[Anim->numKeys];
  for (int i=0; i<Anim->numKeys; i++)
```

```
  {
   Anim->animationKeys[i].Time = *DataPtr;
   if (Anim->animationKeys[i].Time > AnimationSets->Time)
    AnimationSets->Time = Anim->animationKeys[i].Time;
   DataPtr++;
   DataPtr++;

   D3DXMATRIX *tempMatrix = (D3DXMATRIX *)DataPtr;
   DataPtr += 16;

   Anim->animationKeys[i].m_matKey = * tempMatrix;
  }
 }
}
```

Чтобы проще было понимать код, вспомните описание ключа AnimationKey из *разд. 3.12*. В этом блоке мы получим четыре целых числа: тип ключа, количество ключей, время, количество данных и матрицу преобразований. Получением этих данных мы занимаемся в данном блоке кода. Но сначала мы должны создать экземпляр класса CAnimation, в котором будем сохранять загружаемые ключи. После этого получаем данные с помощью метода GetData:

```
pDataObj->GetData(NULL, &TemplateSize, (PVOID*)&DataPtr);
```

Теперь переменная DataPtr указывает на загруженные данные блока Animation. В начале этого блока идет тип ключа, и мы производим проверку, равен ли этот тип четырем (тип ключа 4 соответствует матрице):

```
if (*DataPtr==4)
```

После этой проверки увеличиваем указатель на 4 (4 байта или 2 слова, что соответствует размеру целого значения), чтобы получить указатель на следующее значение, а это количество ключей, которое сохраняем в Anim->numKeys:

```
Anim->numKeys = *DataPtr;
```

Зная количество ключей, выделяем необходимую память для хранения этих данных:

```
Anim->animationKeys = new CAnimationKey[Anim->numKeys];
```

Теперь запускаем цикл, который будет перебирать все ключи. Внутри цикла получаем время ключа и проверяем, если это время больше AnimationSets->Time (времени всей анимации), то сохраняем полученное значение в параметре AnimationSets->Time.

Теперь увеличиваем указатель DataPtr дважды. Почему? Время мы уже получили, а следующий параметр определяет количество данных ключа. Для мат-

рицы это значение всегда равно 16, поэтому можем пропустить этот параметр, а перейти непосредственно к чтению данных матрицы.

3.14. Использование анимации

Теперь мы должны научиться в зависимости от прошедшего времени рассчитать матрицы положения всех костей. Для этого добавим в наш класс управления объектом функцию `UpdateSkinAnimation`, которую вы можете увидеть в листинге 3.13.

Листинг 3.13. Функция обновления

```
void CDXGObject::UpdateSkinAnimation(DWORD Time)
{
 // Если время превышает время анимации, то начать с начала
 if (Time > AnimationSets->Time)
  Time = Time % (AnimationSets->Time);

 // Обновить кости
 CAnimation *Anim = AnimationSets->Animations;
 while (Anim)
 {
   // Обнуляем (делаем единичной) матрицу текущей кости
   D3DXMatrixIdentity(&Anim->Bone->TransformationMatrix);
   // Если ключи существуют, то ...
   if (Anim->numKeys && Anim->animationKeys)
   {
    // Поиск ключей, между которыми находится текущая временная точка
    int Key1 = 0, Key2 = 0;
    for (int i=0; i<Anim->numKeys; i++)
     if (Time >= Anim->animationKeys[i].Time)
      Key1 = i;

    Key2 = (Key1>=(Anim->numKeys-1)) ? Key1 : Key1+1;

    // Рассчитываем матрицу преобразования для данной временной точки
    D3DXMATRIX newMatrix = (Anim->animationKeys[Key2].m_matKey -
        Anim->animationKeys[Key1].m_matKey)*
       (Time - Anim->animationKeys[Key1].Time) /
       (Anim->animationKeys[Key2].Time -
        Anim->animationKeys[Key1].Time);
    newMatrix += Anim->animationKeys[Key1].m_matKey;
```

5 Зак. 543

```
    // Устанавливаем новую точку анимации
    Anim->Bone->TransformationMatrix *= newMatrix;
   }
  // Получаем следующую анимацию для обработки
  Anim = Anim->NextAnim;
 }
}
```

В самом начале функции проверяем — если полученное в качестве параметра время больше времени анимации, то надо разделить текущее время на время анимации.

После этого запускается цикл, который перебирает все анимируемые кости. В этом цикле сначала обнуляем матрицу кости, а потом проверяем, есть ли для косточки ключи анимации. Если есть, то необходимо производить расчеты. Расчеты заключаются в том, чтобы найти два ключа — один с ближайшим меньшим значением времени, а другой с ближайшим большим. Посмотрите на рис. 3.10, где показана временная шкала. Если текущее время находится в указанной точке, то в цикле мы должны найти ключи 3 и 4.

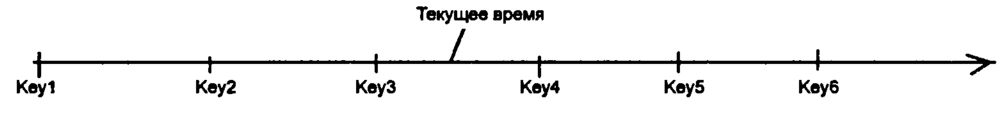


Рис. 3.10. Временная шкала

На основе полученного значения необходимо рассчитать такую матрицу, которая будет соответствовать данной точке. Подобное мы уже делали в *разд. 3.11*. В данном случае код только выглядит сложнее, а на самом деле выполняет абсолютно то же самое.

После завершения цикла получаем указатель на следующую анимацию, чтобы обработать ее:

```
Anim = Anim->NextAnim;
```

Функция `UpdateSkinAnimation` имеет один недостаток, т. е. недоработку — она отображает первый набор анимации `AnimationSets`, но мы же могли загрузить их несколько и необходимо какой-либо способ выбора нужного набора анимации. В файле tiny.x только один набор, поэтому я не даю выбора, чтобы не усложнять код.

Если у вас немного наборов анимации, то можно создать таблицу с указателями на их память и при вызове функции `UpdateSkinAnimation` передавать в качестве параметра еще и указатель на нужный набор анимации. Если вы не знаете, в какой последовательности могут быть наборы анимации в файле, то

придется передавать в функцию имя нужной анимации и искать необходимую в начале с помощью цикла.

Для повышения производительности в играх я рекомендую передавать конкретный указатель. Вы должны четко знать последовательность наборов анимации в файле, чтобы избавиться от лишнего цикла поиска.

Теперь в классе графического движка достаточно добавить следующий код, и пример будет законченным:

```
// Получаем текущее время
double tCurrentTime = GetTickCount();

// Поворачиваем матрицу на 1.5
D3DXMATRIX Rot;
D3DXMatrixRotationX(&Rot, 1.5f);

// Корректируем скелет с учетом прошедшего времени
objects[1]->UpdateSkinAnimation(tCurrentTime-tStartAnimTime);

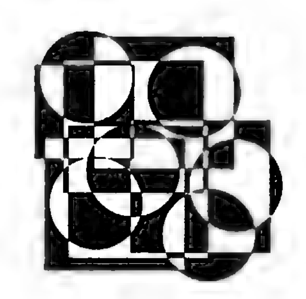
// Обновляем скелет
objects[1]->SetMatrix(&Rot, objects[1]->SkinFrame);
objects[1]->UpdateMesh();
```

Запустите пример и убедитесь, что наш персонаж начал ходить по миру. Как видите, все достаточно просто, и удобно. Вы можете возложить создание анимации профессионалу и спокойно заниматься повышением качества отображения ваших сцен и оптимизацией кода.

Примечание

Исходный код проекта из этой главы можно найти на компакт-диске в каталоге Chapter3\AnimationSkin2.

ГЛАВА 4



Войдите

Игры — это не только графика и анимация. Это программы, которые должны взаимодействовать с пользователем, а значит, необходимо получать ввод данных с клавиатуры, мыши или джойстика. Из-за этого разнообразия устройств может показаться, что взаимодействие с каждым устройством придется программировать отдельно, но библиотека DirectX и ее интерфейс DirectInput прячут от нас все сложности и разнообразия. В этой главе мы рассмотрим работу с устройствами ввода с уклоном на программирование игр, ведь здесь есть свои нюансы.

Если вы еще не успели поработать с этим интерфейсом, то в этой главе мы напишем пример, который может пригодиться в будущем, и внедрим поддержку клавиатуры в наш движок. Для написания примера я использовал некоторые решения, которые реализованы в примере ActionBasic из состава DirectX SDK, потому что это простой пример, а красота и скорость кроются в простоте. Конечно, мне пришлось все переписать заново, с целью оптимизации и подгонки под игровые условия, но идеи схожи.

В составе DirectX SDK есть еще класс `CInputDeviceManager`, который должен упрощать использование интерфейса DirectInput, но мне кажется, что он только усложнил восприятие кода и упрощает только инициализацию, которая реализуется двумя методами. Поэтому мы его использовать не будем, а обратимся к DirectInput напрямую.

4.1. Введение в DirectInput

Интерфейс DirectInput — это программный интерфейс для устройств ввода, таких как мышь, клавиатура, джойстик, руль и другие устройства, которые могли бы использоваться в играх. Имеется возможность использовать этот интерфейс и с устройствами с обратной связью. На данный момент не то, что

джойстик, уже даже мыши есть с обратной связью. Я видел мышь, которая при движении по столу имитировала кочки при наведении на кнопки, но такие устройства не очень прижились. Видимо из-за своей дороговизны и минимальной пользы.

Получается, что при работе с такими устройствами, как мышь и клавиатура (имеются в виду классические варианты), интерфейс представляет собой устройство ввода. Но для некоторых случаев может быть не только ввод, но и вывод, если устройством поддерживается обратная связь.

Почему мы должны использовать интерфейс DirectInput, когда есть классический способ "поймать" события от клавиатуры и мыши? В пользу использования возможностей DirectX можно выделить следующее:

1. Дело в том, что классическая система обработки событий WinAPI слишком медлительна, а DirectInput через прямой доступ к устройству стремится максимально быстро получить результат и вернуть его программе.
2. С помощью WinAPI можно обработать не все устройства ввода. В некоторых случаях может помочь только DirectX.
3. За счет прямого доступа к устройству настройки Windows не влияют на результат, и пример будет работать одинаково на разных настройках, разных компьютеров.
4. Для обращения к специфичным (не стандартным устройствам) приходится работать напрямую с драйвером. Учитывать все возможные драйверы в наше время просто нереально. Вспоминаю времена MS-DOS, когда для полноценной поддержки звука в игре приходилось работать с драйвером напрямую. Если у пользователя звуковая карта не была совместимой с Sound Blaster Pro или совместима, но не полностью, то звук исчезал, и приходилось играть, как в немом кино.
5. Интерфейс DirectInput предоставляет универсальный способ работы с любыми устройствами. Когда мы закончим рассмотрение примера из этой главы, то вы убедитесь, как просто получить доступ к различным устройствам.

Это основные преимущества, но вот третий пункт требует дополнительных комментариев. Дело в том, что настройки ОС нужно игнорировать, но только не для всех устройств. Если проигнорировать настройки джойстика, то пользователь не сможет его откалибровать, и игра будет работать некорректно из-за некорректной калибровки. А вот такие настройки мыши, как скорость перемещения, игнорировать просто необходимо.

Еще одно серьезное отличие DirectInput от классической модели работы с устройствами ввода — если нажать и удерживать клавишу на клавиатуре, то WinAPI через определенные промежутки времени (настраивается в контрольной панели) будет генерировать события об очередном нажатии этой же

клавиши. В DirectInput такого нет. Сколько бы вы не удерживали клавишу, никаких событий не будет. Можно только определить, нажата ли в данный момент клавиша. Это очень удобное отличие, ведь если были бы события, то скорость движения камеры по виртуальному миру зависела бы от частоты посылки таких сообщений. Да и сами сообщения — излишний расход производительности и ненужные прерывания работы программы.

Использование DirectInput не так уж и просто и есть множество проблем, о которых мы еще поговорим по мере их проявления. А сейчас давайте уже знакомиться с интерфейсом DirectInput, и на практике увидим его использование.

4.2. Класс для входа

Для реализации работы с устройствами ввода нам достаточно будет всего трех методов, но лучше выделить их в отдельный класс и отдельный модуль. Методов мало, а описаний будет много, и они просто замусорят движок игры, если мы внедрим все это описание в модуль `CGraphEngine`. Да, код работы с устройствами ввода нельзя назвать красивым. Получается, что класс мы создаем не в целях производительности или удобства, а с целью эстетичности и красоты. Один раз написав модуль, вы забудете про него на долгое время.

Итак, давайте создадим модуль InputCl для хранения необходимого описания и класса `CInputEngine`. Заголовочный файл модуля показан в листинге 4.1.

Листинг 4.1. Заголовочный файл модуля поддержки клавиатуры

```
#ifndef INPUTCL_H
#define INPUTCL_H

#include <dinput.h>

// Уникальный идентификатор для приложения
// {50EB7CDA-FB22-464c-8F96-EED3B5683E9F}
const GUID g_guidApp = { 0x50eb7cda, 0xfb22, 0x464c,
   {0x8f, 0x96, 0xee, 0xd3, 0xb5, 0x68, 0x3e, 0x9f}};

// Перечисление действий, используемых в программе
enum ActionsEnum{
 WALK_LEFT,
 WALK_RIGHT,
 WALK_FORWARD,
 WALK_BACK,
 PUNCH,
```

```
 LOOK_X,
 LOOK_Y,
 QUIT
};

// Константы
const int MAX_DEVICES     = 8;
const int BUTTON_DOWN     = 0x80;
const int NUM_OF_ACTIONS  = 8;

// Структура описания устройства
struct DeviceState
{
 LPDIRECTINPUTDEVICE8 pDevice;
 bool bAxisRelative;
 DWORD dwInput[NUM_OF_ACTIONS];
 bool bMapped[NUM_OF_ACTIONS];
};

// Класс CInputEngine
class CInputEngine
{
private:
public:
 DeviceState aDevices[MAX_DEVICES];
 int iNumDevices;

 CInputEngine();
 int GetActionsState();
 HRESULT AddDevice(const DIDEVICEINSTANCE* lpddi,
        const LPDIRECTINPUTDEVICE8 lpdid);
};

#endif
```

Сразу нужно заметить, что для работы с интерфейсом DirectInput необходимо подключить заголовочный файл dinput.h и в свойствах проекта добавить при сборке использование библиотеки dinput8.lib. Обратите внимание, что мы рассматриваем DirectX9, а DirectInput почему-то восьмой версии. Почему такая несправедливость? С этим вопросом к фирме Microsoft S. Двумерная библиотека не обновлялась уже с 7-й версии, поэтому там надо использовать DirectDraw7.

Теперь давайте посмотрим, что у нас здесь есть. В самом начале в переменной g_guidApp объявляется уникальный идентификатор. Для каждого приложения вам нужно генерировать свой идентификатор. При этом внутри приложения должен использоваться один и тот же идентификатор.

После этого объявляется перечисление ActionsEnum, внутри которого описаны действия, которые мы хотим отлавливать. Эти события никак не связаны с клавишами на клавиатуре или событиями состояния мыши. Вы сами придумываете имена (псевдонимы).

После этого объявляем константы, которые нам понадобятся:

- NUM_OF_ACTIONS — определяет количество действий и должна быть равна количеству элементов в перечислении AgtionsEnum.
- MAX_DEVICES — максимальное количество устройств. В принципе, на компьютере не будет больше 4-х устройств ввода с возможностью использования в играх, но лучше оставить запас и сделать эту константу равной 8;
- BUTTON_DOWN — маска нажатой клавиши. С помощью этой маски мы будем проверять состояние клавиш — нажата она или нет.

Следующим этапом описываем структуру, которая будет использоваться нами для описания устройства. Структура состоит из следующих полей:

- pDevice — указатель на устройство;
- bAxisRelative — флаг, определяющий устройства, имеющие ось, такие как мышь и джойстик;
- dwInput — массив, определяющий состояния зарегистрированных нами действий;
- bMapped — этот массив определяет, является ли соответствующее действие зарегистрированным для данного устройства.

После этого идет описание класса CInputEngine, который будет использоваться для работы с клавишами. Для класса нам понадобится всего два свойства: aDevices (массив типа DeviceState для хранения информации обо всех найденных устройствах ввода) и iNumDevices (количество найденных устройств).

У класса есть еще конструктор, в котором будет происходить инициализация и два метода:

- AddDevice — добавляет информацию о найденном устройстве в массив aDevices и регистрирует клавиши, события которых мы хотим отслеживать;
- GetActionsState — читает информацию с устройств и сохраняет значения в поле dwInput соответствующего устройства. После этого вы можете по значениям массива узнать о состоянии клавиш или мыши.

Как видите, класс получился небольшой. Давайте посмотрим, как реализовать все вышесказанное в виде кода.

4.3. Реализация класса ввода

Начнем рассмотрение реализации с инициализации, т. е. с конструктора `CInputEngine`, который вы можете увидеть в листинге 4.2.

Листинг 4.2. Конструктор класса `CInputEngine`

```
CInputEngine::CInputEngine()
{
 // Обнуляем количество устройств
 iNumDevices = 0;

 // Инициализация DirectInput
 if (FAILED(DirectInput8Create(GetModuleHandle(NULL),
DIRECTINPUT_VERSION,
    IID_IDirectInput8, (VOID**)&diGameInput, NULL)))
  return;

 // Заполнение структуры DIACTIONFORMAT
 ZeroMemory(&diActionFormat, sizeof(DIACTIONFORMAT));
 diActionFormat.dwSize = sizeof(DIACTIONFORMAT);
 diActionFormat.dwActionSize = sizeof(DIACTION);
 diActionFormat.dwDataSize = GetNumOfMappings() * sizeof(DWORD);
 diActionFormat.guidActionMap = g_guidApp;
 diActionFormat.dwGenre = DIVIRTUAL_FIGHTING_HAND2HAND;
 diActionFormat.dwNumActions = GetNumOfMappings();
 diActionFormat.rgoAction = ActionMap;
 diActionFormat.lAxisMin = -99;
 diActionFormat.lAxisMax = 99;
 diActionFormat.dwBufferSize = 16;
 _tcscpy( diActionFormat.tszActionMap, _T("Flenov Game Engine") );

 // Перечислить устройства ввода
 if (FAILED(diGameInput->EnumDevicesBySemantics(NULL, &diActionFormat,
      EnumDevicesCallback, this, DIEDBSFL_ATTACHEDONLY)))
   return;
}
```

Первым делом инициализируем интерфейс `DirectInput` с помощью функции `DirectInput8Create`, которая в общем виде выглядит следующим образом:

```
HRESULT WINAPI DirectInput8Create(
  HINSTANCE hinst,
  DWORD dwVersion,
  REFIID riidltf,
  LPVOID* ppvOut,
  LPUNKNOWN punkOuter
);
```

Метод принимает в качестве параметров:

- hinst — экземпляр приложения, которое создает объект DirectInput;
- dwVersion — версия интерфейса DirectInput. В качестве этого параметра необходимо указать константу DIRECTINPUT_VERSION. Если значение этой константы не объявлено, то будет использоваться значение по умолчанию (0x0800), а во время компиляции даже появится соответствующее сообщение: DIRECTINPUT_VERSION undefined. Defaulting to version 0x0800 (DIRECTINPUT_VERSION не определена. Используется версия по умолчанию 0x0800);
- riidltf — уникальный идентификатор необходимого интерфейса, а для 8-й версии используйте константу IID_IDirectInput8;
- ppvOut — переменная, в которую в случае удачи будет записан созданный интерфейс;
- punkOuter — интерфейс, который должен использоваться для группировки. Если группировка не нужна, то этот параметр должен быть нулевым.

Создав интерфейс DirectInput, нужно перечислить все доступные устройства ввода. Но чтобы это сделать, необходимо подготовить структуру типа DIACTIONFORMAT, которая выглядит следующим образом:

```
typedef struct {
  DWORD                dwSize;
  DWORD                dwActionSize;
  DWORD                dwDataSize;
  DWORD                dwNumActions;
  LPDIACTION           rgoAction;
  GUID                 guidActionMap;
  DWORD                dwGenre;
  DWORD                dwBufferSize;
  OPTIONAL LONG        lAxisMin;
  OPTIONAL LONG        lAxisMax;
  OPTIONAL HINSTANCE hInstString;
  FILETIME             ftTimeStamp;
  DWORD                dwCRC;
  TCHAR                tszActionMap[MAX_PATH];
} DIACTIONFORMAT;
```

Дабы в структуре не было "мусора" в незаполненных явно полях, обнуляем ее с помощью WinAPI-функции ZeroMemory. После этого заполняем поля. Давайте посмотрим, что они означают и для чего нужны:

- dwSize — размер структуры. Это поле должно быть обязательно заполнено и заполнено корректно, а нам не сложно определить размер с помощью функции sizeof;
- dwActionSize — размер структуры DIACTION, которая отражает одно игровое действие на одно игровое устройство;
- dwDataSize — определяет размер данных и должно быть равно значению параметра dwDataSize, умноженное на 4;
- dwNumActions — количество действий;
- rgoAction — указатель на массив структур DIACTION, содержащий необходимые нам игровые действия;
- guidActionMap — уникальный идентификатор приложения, который мы сгенерировали в заголовочном файле;
- dwGenre — определяет жанр приложения. В этом параметре может быть одна из констант, например, DIVIRTUAL_FIGHTING_HAND2HAND;
- dwBufferSize — определяет размер буфера входящих пакетов данных для каждого устройства, для которого установлена карта действий;
- lAxisMin — минимальное значение, которое может быть возвращено устройствами типа "мышь" или "джойстик" во время движения вдоль какой-либо оси. Клавиши на клавиатуре и кнопки на том же джойстике не имеют оси, поэтому этот параметр не используют:
  - lAxisMax — максимальное значение, возвращаемое осевым устройством ввода;
  - hInstString — идентификатор модуля, содержащий строковые ресурсы для всех действий;
  - ftTimeStamp — структура FILETIME, которая получает время, когда карта действий последний раз записывалась на диск;
  - dwCRC — значение контрольной суммы для данной карты;
  - tszActionMap — текстовая строка, которая содержит понятное описание для данной карты действий.

Наиболее интересным свойством является параметр rgoAction, в котором должен быть массив из действий. В примере из листинга 4.2 в этом параметре указана переменная ActionMap, которая как раз и описывает действия. Если с остальными параметрами все понятно, то этот необходимо пояснить. Так давайте же воочию увидим это объявление:

```
DIACTION ActionMap[] =
{
 // События клавиатуры для перемещения
 {WALK_LEFT,    DIKEYBOARD_LEFT,  0, ACTION_NAMES[WALK_LEFT], },
 {WALK_RIGHT,   DIKEYBOARD_RIGHT, 0, ACTION_NAMES[WALK_RIGHT], },
 {WALK_FORWARD, DIKEYBOARD_UP,    0, ACTION_NAMES[WALK_FORWARD],},
 {WALK_BACK,    DIKEYBOARD_DOWN,  0, ACTION_NAMES[WALK_BACK], },

 // События клавиатуры для удара и выхода из программы
 {PUNCH, DIKEYBOARD_P, DIA_APPFIXED, ACTION_NAMES[PUNCH], },
 {QUIT,  DIKEYBOARD_Q, DIA_APPFIXED, ACTION_NAMES[QUIT], },

 // События мыши
 {LOOK_X, DIMOUSE_XAXIS,   0,  ACTION_NAMES[LOOK_X], },
 {LOOK_Y, DIMOUSE_YAXIS,   0,  ACTION_NAMES[LOOK_Y], },
 {PUNCH,  DIMOUSE_BUTTON0, 0,  ACTION_NAMES[PUNCH], },
};
```

Как видите, переменная ActionMap является массивом из структур типа DIACTION. Каждая структура описывает одно действие и связывает его с определенным событием устройства. При описании структуры, мы заполняем первые четыре параметра (в реальности их больше).

Первый параметр — это константа из перечисления enum, которое мы описали в заголовочном файле под именем ActionsEnum. Через этот параметр мы указываем имя для регистрируемого действия.

Второй параметр — это действие устройства. Константы для клавиатуры начинаются с DIKEYBOARD_, джойстика с JOY_, а для мыши — с префикса DIMOUSE_. После этого все просто, необходимо только добавить к префиксу имя кнопки или букву на клавише клавиатуры. Если необходима ось устройства, то к префиксу нужно добавить XAXIS или YAXIS. Я думаю, что эта подсказка позволит мне не описывать константы для всех клавиш клавиатуры, мыши и джойстика, потому что их больше 100 (одних клавиш на клавиатуре больше 100), и вы сами сможете определить необходимое вам имя константы. Получается, что вторым параметром мы указываем, какая клавиша должна генерировать действие, описанное в первом параметре.

Третий параметр структуры DIACTION — это флаги. Для всех действий мы устанавливаем нулевой флаг, кроме клавиш клавиатуры, где указывается DIA_APPFIXED. Этот флаг указывает на то, что действие не может быть переопределено с помощью Microsoft DirectInput.

Последний параметр — это текстовое описание команды. Это описание может отображаться в окне конфигурирования действий, если вы захотите дать пользователю возможность самому устанавливать клавиши.

4.4. Перечисление устройств

После заполнения структуры DIACTIONFORMAT корректными "знамениями" мы вызываем функцию EnumDevicesBySemantics, которая запускает перечисление всех устройств ввода, которые точно подходят к указанной карте действий. Этот метод в общем виде выглядит следующим образом:

```
HRESULT EnumDevicesBySemantics(
  LPCTSTR ptszUserName,
  LPDIACTIONFORMAT lpdiActionFormat,
  LPDIENUMDEVICESBYSEMANTICSCB lpCallback,
  LPVOID pvRef,
  DWORD dwFlags
);
```

Здесь у нас 5 параметров:

- ptszUserName — строка, содержащая имя пользователя. Установите нулевое значение, чтобы использовать имя, под которым вы вошли в систему;
- lpdiActionFormat — указатель на корректно заполненную структуру DIACTIONFORMAT;
- lpCallback — указатель на функцию обратного вызова, которая будет вызвана для каждого найденного устройства;
- pvRef — дополнительный параметр, который будет передан в функцию перебора устройств. Через этот параметр мы будем передавать указатель на текущий класс, чтобы через эту переменную получать доступ к полям класса;
- dwFlags — флаги, определяющие перечисление. Флагов очень много и за полным списком следует обратиться к документации, а мы будем использовать флаг DIEDBSFL_ATTACHEDONLY. Перечислять только подключенные к компьютеру и проинсталлированные устройства. Я думаю, что вы тоже его будете использовать в большинстве своих приложений.

В данном случае третий параметр требует пояснений. Давайте посмотрим, что это за функция, как она будет ловить и сохранять найденные устройства. В нашем приложении эта функция будет очень простой, потому что самое интересное мы спрячем в другую функцию, но об этом в свое время. Давайте сейчас посмотрим на EnumDevicesCallback:

```
BOOL CALLBACK EnumDevicesCallback(LPCDIDEVICEINSTANCE lpddi,
      LPDIRECTINPUTDEVICE8 lpdid, DWORD dwFlags,
      DWORD dwRemaining, LPVOID pvRef)
```

```
{
 ((CInputEngine*)pvRef)->AddDevice(lpddi, lpdid);

 return DIENUM_CONTINUE;
}
```

Через параметр pvRef мы получаем значение, которое было указано в одноименном и четвертом по счету параметре функции EnumDevicesBySemantics. Мы передавали здесь указатель на текущий объект и через эту переменную мы вызываем метод AddDevice нашего класса, который сделает всю необходимую работу. В качестве результата функция должна вернуть константу DIENUM_CONTINUE, чтобы перечисление устройств продолжилось.

Итак, в листинге 4.3 показан метод AddDevice.

Листинг 4.3. Метод AddDevice, который сохраняет информацию об устройстве

```
HRESULT CInputEngine::AddDevice(const DIDEVICEINSTANCE* lpddi,
                                const LPDIRECTINPUTDEVICE8 lpdid)
{
 HRESULT hr;

 // Вышли ли мы за пределы
 if (iNumDevices < MAX_DEVICES)
 {
  // Связать карту действий с устройством
  if (FAILED(hr=lpdid->BuildActionMap(&diActionFormat,
             NULL, DIDBAM_DEFAULT)))
   return DIENUM_CONTINUE;

  // Цикл проверки, установлено ли действие
  for (UINT i=0; i < diActionFormat.dwNumActions; i++)
  {
   DIACTION *dia = &(diActionFormat.rgoAction[i]);

   if (dia->dwHow != DIAH_ERROR && dia->dwHow != DIAH_UNMAPPED)
    aDevices[iNumDevices].bMapped[dia->uAppData] = TRUE;
  }
  // Установить формат данных для устройства и отразить
  // определенные приложением действия на объект устройства
  if (FAILED(hr = lpdid->SetActionMap(&diActionFormat,
     NULL, DIDSAM_DEFAULT)))
  {
   ZeroMemory(aDevices[iNumDevices].bMapped,
           sizeof(aDevices[iNumDevices].bMapped) );
```

```
    return DIENUM_CONTINUE;
  }

  aDevices[iNumDevices].pDevice = lpdid;
  lpdid->AddRef();

  // Заполнение структуры DIPROPDWORD
  DIPROPDWORD dipdw;
  dipdw.diph.dwSize = sizeof(DIPROPDWORD);
  dipdw.diph.dwHeaderSize = sizeof(DIPROPHEADER);
  dipdw.diph.dwObj = 0;
  dipdw.diph.dwHow = DIPH_DEVICE;

  // Получение данных
  hr = lpdid->GetProperty(DIPROP_AXISMODE, &dipdw.diph);
  if (SUCCEEDED(hr))
   aDevices[iNumDevices].bAxisRelative =
          (DIPROPAXISMODE_REL == dipdw.dwData);

  g_bRefChart = TRUE;
  iNumDevices++;
 }

 // Возвращаем DIENUM_CONTINUE, чтобы продолжить перечисление
 return DIENUM_CONTINUE;
}
```

Просто увидеть код — мало. Если вы никогда не работали с интерфейсом DirectInput, то код будет как будто написан на китайском.

В качестве первого параметра метод AddDevice получает структуру DIDEVICEINSTANCE, которая описывает найденное устройство. Эта структура выглядит следующим образом:

```
typedef struct DIDEVICEINSTANCE {
  DWORD dwSize;
  GUID  guidInstance;
  GUID  guidProduct;
  DWORD dwDevType;
  TCHAR tszInstanceName[MAX_PATH];
  TCHAR tszProductName[MAX_PATH];
  GUID  guidFFDriver;
  WORD  wUsagePage;
  WORD  wUsage;
} DIDEVICEINSTANCE, *LPDIDEVICEINSTANCE;
```

Кратко пробежимся по полям этой структуры:

- dwSize — размер структуры;
- guidInstance — уникальный идентификатор экземпляра устройства;
- guidProduct — глобальный идентификатор продукта, который устанавливается производителем;
- dwDevType — тип устройства;
- tszInstanceName — дружественное имя устройства, например "клавиатура";
- tszProductName — дружественное имя продукта;
- guidFFDriver — уникальный идентификатор для драйвера, используемый для обратного вызова;

Второй параметр, который мы получаем в функции AddDevice, — указатель на интерфейс IDirectInputDevice8. Это интерфейс для работы с устройством и через него мы будем регистрировать наши действия. Именно это и происходит сразу после проверки, не вышли ли мы за пределы максимального количества устройств. Под хранение информации об устройствах мы выделили массив из 10 элементов, но чем черт не шутит, вдруг мы выскочим за пределы массива и лучше добавить эту проверку.

Итак, для начала необходимо построить карту действия для данного устройства и получить информацию о ней. Это происходит с помощью метода BuildActionMap интерфейса IDirectInputDevice8:

```
HRESULT BuildActionMap(
 LPDIACTIONFORMAT lpdiaf,
 LPCTSTR lpszUserName,
 DWORD dwFlags
);
```

Здесь имеется три параметра:

- указатель на структуру DIACTIONFORMAT, которую мы заполнили на этапе инициализации в конструкторе *(см. разд. 4.3)*;
- строка, содержащая имя пользователя, если ноль, то будет использоваться текущий пользователь;
- флаги.

В качестве последнего параметра в нашем примере в качестве флага используется DIDBAM_DEFAULT, при котором перезаписываются все карты, за исключением назначенных приложением.

Если работа метода завершилась ошибкой, то продолжать установку действий нет смысла. Просто возвращаем DIENUM_CONTINUE, чтобы продолжить пе-

речисление устройств и попробовать установить карту действий другим устройствам.

После этого запускаем цикл, в котором проверяем, является ли действие доступное этому устройству:

```
for (UINT i=0; i < diActionFormat.dwNumActions; i++)
{
 // Получить указатель на очередное действие
 DIACTION *dia = &(diActionFormat.rgoAction[i]);

 // Если свойство dwHow не равно ошибке,
 // то действие может быть назначено
 if(dia->dwHow != DIAH_ERROR && dia->dwHow != DIAH_UNMAPPED )
  aDevices[iNumDevices].bMapped[dia->uAppData] = TRUE;
}
```

Теперь необходимо установить формат данных для устройства и отразить определенные приложением действия на объект устройства. Это делается с помощью метода `SetActionMap`:

```
HRESULT SetActionMap(
 LPCDIACTIONFORMAT lpdiActionFormat,
 LPCTSTR lptszUserName,
 DWORD dwFlags
);
```

Тут у нас имеется три параметра. Они вам ничего не напоминают? Да, параметры такие же, как и у метода `BuildActionMap`.

Самое последнее, что мы делаем — определяем, позволяет ли устройство выполнять работу с осями координат. Наиболее популярными такими устройствами являются мышь и джойстик. Например, при движении мышью мы двигаем ею по столу и драйвер получает от устройства изменения значений по осям X и Y (движения мыши двумерны, поэтому две оси). То же самое с джойстиком, у которого изменения значений по осям зависят от наклона рукоятки. Клавиатуру двигать по столу бесполезно, драйвер этого не увидит.

Для определения — регистрирует ли устройство движения по осям, используем метод `GetProperty`, который возвращает значение указанного свойства.

4.5. Опрос состояния действий

Теперь наш класс умеет инициализировать и регистрировать карту действий. Но этой карте грош цена, потому что необходимо уметь определить состояние действий — нажаты ли клавиши на клавиатуре и произошло ли движение

мышью/джойстиком. Именно ради этого мы написали столько кода. В листинге 4.4 вы можете увидеть функцию `GetActionsState`, которая производит опрос всех устройств и всех зарегистрированных действий.

Листинг 4.4. Опрос состояний действий

```
int CInputEngine::GetActionsState()
{
 int r;
 HRESULT hr;

 // Цикл опроса всех устройств
 for (int iDevice=0; iDevice<iNumDevices; iDevice++)
 {
  LPDIRECTINPUTDEVICE8 pdidDevice = aDevices[iDevice].pDevice;
  DIDEVICEOBJECTDATA rgdod[10];

  DWORD dwItems = 10;

  // Получаем доступ к устройству ввода и опрашиваем его
  pdidDevice->Acquire();
  pdidDevice->Poll();
  if (FAILED(pdidDevice->GetDeviceData(sizeof(DIDEVICEOBJECTDATA),
       gdod, &dwItems, 0)));
   continue;

  // Если устройство поддерживает оси,
  // то обнуляем параметры LOOK_X и LOOK_Y
  if (aDevices[iDevice].bAxisRelative)
  {
   aDevices[iDevice].dwInput[LOOK_X] = 0;
   aDevices[iDevice].dwInput[LOOK_Y] = 0;
  }

  // Цикл опроса всех действий
  for( DWORD j=0; j<dwItems; j++ )
  {
   UINT_PTR dwAction = rgdod[j].uAppData;
   int dwData = 0;

   // Определяем тип действия
   switch(dwAction)
   {
    case LOOK_X:
    case LOOK_Y:
```

```
    // Если произошло осевое движение, то определяем,
    // в какую именно сторону смещение джойстика или мыши
    {
     dwData = rgdod[j].dwData;

     if (aDevices[iDevice].bAxisRelative)
     {
      if( (int)dwData < 0 )
       dwData = max((int)dwData, diActionFormat.lAxisMin);
      else
       dwData = min((int)dwData, diActionFormat.lAxisMax);
     }
     break;
    }
    // Действия по умолчанию (для кнопок)
    default:
    {
     // Проверяем, нажата или отпущена
     if (rgdod[j].dwData & BUTTON_DOWN)
      dwData = 1;
     else
      dwData = 0;
     break;
    }
   }

   aDevices[iDevice].dwInput[dwAction] = (DWORD)dwData;
   r++;
  }
 }
return r;
}
```

Теперь класс можно считать законченным. Осталось только разобраться, как происходит сам опрос.

Для опроса состояния клавиш можно поступить двумя способами:

- метод может возвращать состояние определенной клавиши. В этом случае, при каждом вызове метода приходится обращаться к устройству, что отнимает лишние ресурсы;
- один раз в цикле прочитать состояния всех действий в устройстве, занести результат в массив и потом основной движок может одним проходом проверить состояние клавиш без обращения к устройству.

Первый метод плох тем, что приходится постоянно обращаться к устройству, а второй плох тем, что содержит много циклов. С точки зрения производительности, второй вариант быстрее (цикл, но зато нет лишних обращений к устройству, что очень накладно), поэтому я реализовал его.

Итак, в функции GetActionsState сначала запускаем цикл, который перебирает все найденные нами устройства. Внутри цикла сначала получаем доступ к устройству ввода с помощью метода Acquire. Потом опрашиваем данные с объекта с помощью метода Poll. Теперь осталось только прочитать данные из буфера с помощью метода GetDeviceData. В общем виде метод выглядит следующим образом:

```
HRESULT GetDeviceData(
  DWORD cbObjectData,
  LPDIDEVICEOBJECTDATA rgdod,
  LPDWORD pdwInOut,
  DWORD dwFlags
);
```

У этого метода четыре параметра:

- размер структуры DIDEVICEOBJECTDATA;
- указатель на массив структур DIDEVICEOBJECTDATA, через который мы получим данные;
- количество элементов в массиве структур, из второго параметра. После выполнения метода в этом параметре будет реальное количество полученных структур;
- дополнительные флаги. Мы не будем их использовать, а поэтому параметр будет равен нулю.

После получения данных запускаем цикл, в котором проверяем каждое действие. В этом цикле для осевых действий мы должны проверить, в какую сторону произошло движение. Для оси X и Y, если параметр uAppData структуры DIDEVICEOBJECTDATA отрицателен, то движение произошло в отрицательную сторону, иначе в положительную.

Для кнопочных действий мы должны проверить, что произошло — нажатие или отпускание кнопки. Для этого проверяем действие с маской BUTTON_DOWN:

```
if (rgdod[j].dwData & BUTTON_DOWN)
 dwData = 1;
else
 dwData = 0;
```

Результат сохраняется в массиве dwInput. Вот и все.

4.6. Использование класса клавиатуры

Теперь посмотрим, как можно использовать наш класс клавиатуры. Для этого в движке игры сначала объявим следующие переменные:

```
CInputEngine *ieEngine;              // Переменная класса ввода
D3DMATRIX worldMat;                  // Матрица положения камеры
D3DMATRIX xrMat, yrMat, mvMat;       // Матрицы преобразования
D3DXVECTOR3 pos, vecLR, vecFB, up;   // Векторы положения
```

Теперь где-нибудь на этапе инициализации создаем класс работы с устройствами ввода следующим образом:

```
ieEngine = new CInputEngine();
```

Все, можно переходить к опросу устройств. Это лучше сделать перед отображением сцены или по таймеру. Я выбрал функцию отображения. Только чтобы ее не загромождать, выделим работу с клавиатурой в отдельный модуль. Для этого в начале функции `RenderScene` добавим следующие две строки:

```
if (ieEngine->GetActionsState()!=0)
  KeyControl();
```

В операторе `if` мы проверяем, есть ли какие-то действия. Если да, то вызывается функция `KeyControl`, которую можно увидеть в листинге 4.5. В этой функции будет обработка нажатых клавиш и перемещения мыши, а также установка матрицы просмотра VIEW. Из функции `RenderScene` установка матрицы VIEW должна быть убрана.

Листинг 4.5. Функция обработки действий устройств ввода

```
void CGraphEngine::KeyControl()
{
 // Цикл проверки всех структур устройств
 for (int iDevice=0; iDevice<ieEngine->iNumDevices; iDevice++)
 {
  // Нажата ли кнопка Q
  if (ieEngine->aDevices[iDevice].dwInput[QUIT]==1)
   PostQuitMessage(1);

  // Нажата ли стрелка движения вперед
  if (ieEngine->aDevices[iDevice].dwInput[WALK_FORWARD]==1)
   MoveCamera(.1f, &vecFB);

  // Нажата ли стрелка движения назад
  if (ieEngine->aDevices[iDevice].dwInput[WALK_BACK]==1)
   MoveCamera(-.1f, &vecFB);
```

```
  // Нажата ли стрелка движения влево
  if (ieEngine->aDevices[iDevice].dwInput[WALK_LEFT]==1)
   MoveCamera(.1f, &vecLR);

  // Нажата ли стрелка движения вправо
  if (ieEngine->aDevices[iDevice].dwInput[WALK_RIGHT]==1)
   MoveCamera(-.1f, &vecLR);

  // Мышь или джойстик сдвинуты по оси X
  if (ieEngine->aDevices[iDevice].dwInput[LOOK_X]!=0xCDCDCDCD)
  {
   // Сдвиг вправо
   if ((int)ieEngine->aDevices[iDevice].dwInput[LOOK_X]>0)
    RotateCamera(0.05f, 0);

   // Сдвиг влево
   if ((int)ieEngine->aDevices[iDevice].dwInput[LOOK_X]<0)
    RotateCamera(-0.05f, 0);
   ieEngine->aDevices[iDevice].dwInput[LOOK_X]=0;
  }

  // Мышь или джойстик сдвинуты по оси Y
  if (ieEngine->aDevices[iDevice].dwInput[LOOK_Y]!=0xCDCDCDCD)
  {
   // Сдвиг вперед
   if ((int)ieEngine->aDevices[iDevice].dwInput[LOOK_Y]>0)
    RotateCamera(-0.05f, 1);

   // Сдвиг назад
   if ((int)ieEngine->aDevices[iDevice].dwInput[LOOK_Y]<0)
    RotateCamera(0.05f, 1);
   ieEngine->aDevices[iDevice].dwInput[LOOK_Y]=0;
  }
}

// Устанавливаем векторы положений
pos = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
vecLR = D3DXVECTOR3(10.0f, 0.0f, 0.0f);
vecFB = D3DXVECTOR3(0.0f, 0.0f, -10.0f);
up = D3DXVECTOR3(0.0f, 10.0f, 0.0f);

// Перемножаем матрицы
D3DMATRIX View=worldMat;
D3DXMatrixMultiply((D3DXMATRIX*)&View, (D3DXMATRIX*)&xrMat,
    (D3DXMATRIX*)&yrMat);
```

```
  D3DXMatrixMultiply((D3DXMATRIX*)&View, (D3DXMATRIX*)&View,
     (D3DXMATRIX*)&mvMat);

  // Трансформируем координаты
  D3DXVec3TransformCoord(&pos, &pos, (D3DXMATRIX*)&View);
  D3DXVec3TransformCoord(&vecFB, &vecFB, (D3DXMATRIX*)&View);
  D3DXVec3TransformCoord(&vecLR, &vecLR, (D3DXMATRIX*)&View);
  D3DXVec3TransformNormal(&up, &up, (D3DXMATRIX*)&View);

  // Устанавливаем положение камеры
  D3DXMatrixLookAtLH((D3DXMATRIX*)&View, &pos, &vecFB, &up);
  pDevice->SetTransform(D3DTS_VIEW, &View);
}
```

В функции KeyControl запускается цикл перебора всех структур зарегистрированных устройств. Внутри цикла проверяем все действия. Помните, в *разд. 4.5*, в функции GetActionsState мы заполняли в структуре элементы массива dwInput, соответствующие действию. Здесь мы проверяем уже установленные значения. Для этого используется следующая конструкция:

```
ieEngine->aDevices[iDevice].dwInput[действие]
```

Например, чтобы проверить, нажата ли кнопка Quit, необходимо выполнить следующую проверку:

```
if (ieEngine->aDevices[iDevice].dwInput[QUIT]==1)
```

Таким образом, мы проверяем все установленные действия, кроме PUNCH (удар рукой). Это действие я оставил незатронутым, чтобы вы сами попробовали написать для него код. Таким образом, вы закрепите уже пройденный в этой и предыдущих главах материал. По подробным комментариям вы легко разберетесь, где и какое действие обрабатывается. Тут нужно только дать пояснение, как происходит движение и поворот сцены.

Для того чтобы производить движение, используется функция MoveCamera. Ей передается два значения — смещение, на которое нужно переместиться, и вектор, определяющий движение.

Для движения вперед или назад мы передаем вектор из переменной vecFB, который равен (D3DXVECTOR3(0.0f, 0.0f, -10.0f)). Тут может возникнуть вопрос — почему движение идет вдоль оси Z, ведь вектор направлен именно вдоль этой оси? Ответ вполне логичен — ось Z направлена вглубь экрана и когда мы движемся вперед, то мы идем именно вглубь этого экрана. В зависимости от левосторонней или правосторонней системы координат (в какую сторону направлена ось — внутрь экрана или от него) изменяется знак числа, указанного в компоненте Z.

Для движения vecLR, влево и вправо используется вектор, который равен D3DXVECTOR3(10.0f, 0.0f, 0.0f). Этот вектор вытянут вдоль оси X. Ось X всегда направлена вправо, поэтому тут необходимо просто указать в этой составляющей положительное число. В данном случае выбрано число 10.

В наших расчетах используется еще один важный вектор — up, который равен D3DXVECTOR3(0.0f, 10.0f, 0.0f). Этот вектор определяет, куда смотрит макушка персонажа. У нас он стоит и не наклоняется, поэтому вектор направлен вдоль оси Y ровно вверх.

Теперь посмотрим на саму функцию передвижения:

```
void CGraphEngine::MoveCamera(float fOffset,
   D3DXVECTOR3 *vecModifier)
{
 D3DXMATRIX tView;
 D3DXVECTOR3 vect;

 // Вычитаем векторы
 D3DXVec3Subtract(&vect, vecModifier, &pos);
 // Сдвигаем вектор
 vect *= fOffset;

 // Корректируем матрицу положения с учетом рассчитанного вектора
 D3DXMatrixTranslation(&tView, vect.x, vect.y, vect.z);

 // Перемножаем результат корректировки с матрицей движения mvMat
 D3DXMatrixMultiply((D3DXMATRIX*)&mvMat, (D3DXMATRIX*)&mvMat,
    (D3DXMATRIX*)&tView);
}
```

Функция рассчитывает новое положение вектора, смещая его на указанное число, затем создается матрица перемещения на основе рассчитанного вектора, которая перемножается с матрицей движения — mvMat. Таким образом, мы получили одну из необходимых для отображения сцены матриц — перемещение.

Если произошло движение мышью, то необходимо повернуть камеру. Тут тоже свои подводные камни и для вращения мы вызываем функцию RotateCamera:

```
void CGraphEngine::RotateCamera(float fOffset, int Axis)
{
 D3DXMATRIX tView;

 // Вращение вдоль оси Y
 if (Axis==0)
```

```
 {
  D3DXMatrixRotationY(&tView, fOffset);
  D3DXMatrixMultiply((D3DXMATRIX*)&yrMat, (D3DXMATRIX*)&yrMat,
     (D3DXMATRIX*)&tView);
 }

 // Вращение вдоль оси X
 if (Axis==1)
 {
  D3DXMatrixRotationX(&tView, fOffset);
  D3DXMatrixMultiply((D3DXMATRIX*)&xrMat, (D3DXMATRIX*)&xrMat,
    (D3DXMATRIX*)&tView);
 }
}
```

В качестве параметров функция получает угол, на который нужно повернуться, и ось, вдоль которой нужно вертеться. Если ось равна 0, то поворачиваемся вдоль оси Y (по горизонтали) и сохраняем результат в переменной yMat. Это мы получили вторую из необходимых матриц. Третья матрица получается, если требуется повернуться вдоль оси X, т. е. поднять или опустить голову. Это преобразование сохраняется в матрице xrMat.

Теперь, когда у нас есть все три матрицы, мы их перемножаем в функции KeyControl, чтобы получить результирующее положение с учетом перемещения и вращения вдоль обеих осей:

```
D3DMATRIX View=worldMat;
D3DXMatrixMultiply((D3DXMATRIX*)&View, (D3DXMATRIX*)&xrMat,
    (D3DXMATRIX*)&yrMat);
D3DXMatrixMultiply((D3DXMATRIX*)&View, (D3DXMATRIX*)&View,
    (D3DXMATRIX*)&mvMat);
```

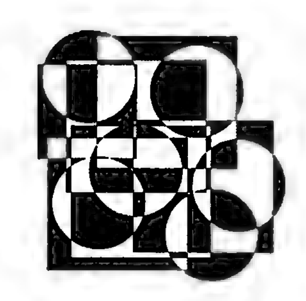
Если ваш персонаж должен еще и уметь наклонять голову (вращение вдоль оси Z), то нужно будет перемножать четыре матрицы, а необходимое вращение добавить в функцию RotateCamera.

Вот и все. Запустите пример и попробуйте побегать по комнате. Только будьте осторожны, в этой игре еще нет проверки на столкновения, и вы без проблем можете пройти сквозь стену и увидеть, что находится за ней.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter4\Input.

ГЛАВА 5



"Фейсом об тейбл"

В *главе 4* мы наделили наш движок возможностью перемещения по виртуальному миру. Но при этом, на данный момент мы можем даже без взмаха волшебной палочки пройти сквозь стену. Это не есть хорошо.

Редкая игра обходится без системы обнаружения коллизий. Даже когда кого-то бьют лицом об стол (именно это выражение дало имя главе), происходит столкновение двух объектов, правда в боевиках в такие моменты стол иногда ломается. В реальной жизни вероятность поломки стола стремится к нулю, разве что стол из картона и на него упадет Турчинский ☺.

Итак, в этой главе мы продолжим совершенствовать наш движок игры и наделим его системой обнаружения столкновений, т. е. теперь наша камера не должна будет так легко проходить сквозь стену или любой другой твердый предмет. Итак, давайте начнем с теории, которой будет немало, а потом на практике рассмотрим подводные камни обнаружения коллизий.

5.1. Алгоритмы обнаружения

Алгоритмов проверки на столкновение множество и об этом можно писать целую книгу. Однажды я даже видел на форуме сайта **www.gemadev.com**, посвященному программированию игр, ссылку на то, что такая книга существует. Я по магазинам хожу редко (в последнее время я и из дома выхожу не часто, потому что работаю, как говорится at home) и на русском такую книгу не видел. В американских магазинах я вообще не бывал, не считая **www.amazon.com**, на котором тоже эта книга мне на глаза не попадалась, а почитал бы я ее с огромным удовольствием.

Существует множество различных методов определения столкновений или пересечений объектов. Самый простейший метод определения столкновения

основан на цветах, но использовался в 2D-играх. Если при наложении слоя оказывается, что на поверхности уже есть точка определенного цвета, принадлежащая спрайту, то произошло столкновение. Если при наложении слоя мы видим только цвет фона, то столкновения нет.

Наш виртуальный мир сложнее, потому что находится в 3D-пространстве, которое неограниченно по трем осям. Заполнять это пространство каким-либо цветом только для того, чтобы определить столкновение — глупо и неэффективно. Проверять такой мир по точкам будет утомительно даже мощной графической станции с несколькими процессорами, поэтому перейдем к математическим методам, которые намного быстрее и позволяют решить проблему коллизий в нашем бесконечном 3D-пространстве.

Давайте посмотрим, как можно решить нашу задачу. Например, стену можно воспринимать как плоскость. Чтобы узнать, произошло ли столкновение со стеной, достаточно проверить, пересекла ли точка плоскость. Да, стена может быть намного сложнее плоскости, но алгоритм определения данного столкновения очень прост и экономичен для процессора. Поэтому мы начнем рассмотрение алгоритмов с пересечения точек и плоскости. На его основе можно сделать более сложные алгоритмы.

5.1.1. Точка против плоскости

Что такое наша камера? В самом примитивном варианте — это точка. Что такое стена? В простейшем варианте — это плоскость. Если бы все было так красиво, то расчет столкновения происходил бы следующим образом:

```
D3DXPLANE plane = плоскость;
D3DVECTOR point = позиция точки;

D3DVECTOR P = D3DXVECTOR(plane.a, plane.b, plane.c);
float Dist = D3DXVec3Dot(&point, &P)+ plane.d;
```

Если расстояние равно нулю, то точка на поверхности. Если расстояние отрицательное, то точка пересекла поверхность. Равенства добиться сложно, его нужно только рассчитывать, ведь при определении положения в 3D-мире мы оперируем вещественными числами. Двигаясь по миру, очень сложно попасть точно на поверхность, поэтому тут происходят округления, и точка считается на поверхности, если расстояние не равно нулю, но очень близко к нему.

5.1.2. Точка против куба

Самое простейшее пересечение — это точка с кубом, точнее с ящиком/боксом. Оно даже проще, чем проверка столкновения с плоскостью. *Куб* — это

фигура, в которой все грани одинаковой длины, а мы рассматриваем общий случай, в котором стороны могут быть разной длины, т. е. грань выглядит в виде прямоугольника, а не квадрата. Тем более что разницы в проверке столкновения с боксом и кубом нет.

Для проверки входа точки в бокс нам необходимы только координата точки и две координаты, описывающие бокс: минимальная нижняя точка и максимальная верхняя точка. Имея эти данные, можно проверить вхождение следующим образом:

```
if (point.x > BoundingBoxMin.x &&
    point.x < BoundingBoxMax.x &&
    point.y > BoundingBoxMin.y &&
    point.y < BoundingBoxMax.y &&
    point.z > BoundingBoxMin.z &&
    point.z < BoundingBoxMax.z)
```

А что, если необходимо посчитать расстояние от точки до куба? В этом случае можно произвести сравнение точки с плоскостью необходимой стороны куба. Если нужно определить расстояние до центра точки, то просто заключите куб в сферу и рассчитывайте расстояние до сферы. Но это уже отдельная тема, которая рассмотрена в *разд. 5.1.3*.

5.1.3. Точка против сферы

Теперь посмотрим, как можно проверить пересечение точки сферы. На первый взгляд это не очень простая задача, потому что поверхность сферы не плоская. Но это проблема только на первый взгляд, тут необходимо лишь рассчитать расстояние от точки до центра сферы. Если расстояние равно радиусу, то точка на поверхности, а если меньше радиуса, то точка уже внутри сферы.

В виде кода определение пересечения может выглядеть так:

```
D3DVECTOR sphere = позиция точки;
int radius = радиус;
D3DVECTOR point = позиция точки;

float DistVector = sphere - point;
if (sqrt(DistVector.x * DistVector.x+
         DistVector.y * DistVector.y +
         DistVector.z * DistVector.z)<Radius)
 // Точка пересекла сферу
```

В этом примере есть один недостаток — при расчете расстояния приходится рассчитывать квадратный корень, а это достаточно накладно для процессора.

Но проблему можно решить так, чтобы не рассчитывать корень, а просто возвести радиус в квадрат — результат будет один и тот же:

```
if (DistVector.x * DistVector.x+
        DistVector.y * DistVector.y +
        DistVector.z * DistVector.z < Radius * Radius)
  // Точка пересекла сферу
```

5.1.4. Сфера против сферы

Попробуйте сами сейчас подумать, как можно определить столкновение двух сфер? Все достаточно просто, необходимо только рассчитать расстояние между центрами двух сфер, и если оно меньше, чем сумма двух радиусов, то столкновение произошло. Попробуйте сами написать необходимую формулу.

5.2. Столкновения в играх

Об определении коллизий можно говорить долго и нудно, потому что существует множество геометрических фигур и между ними коллизии проверяются по-разному. Но проверка столкновений в 3D-программах и в играх — это разные вещи. Чтобы с высокой точностью узнать, произошло ли столкновение у двух 3D-объектов, необходимы немалые ресурсы. В графических пакетах, где нет необходимости отображать сцену в реальном времени, мы можем рассчитывать пересечения с высокой точностью, но в играх такие расходы просто непозволительны.

Дабы добиться расчетов в реальном времени, мы можем "мухлевать" и упрощать расчеты, и давайте посмотрим, как это делать. Для начала посмотрим, с чем именно мы можем мухлевать, и как добиться хоть немного, приближенного к реальности результата.

Рассуждать о столкновениях с точки зрения ящиков и сфер достаточно просто, но в нашей игре с ящиком можно сравнить только стены комнаты. В реальной игре даже стены будут не квадратными. Персонаж, который мы поместили в комнате не то, что с квадратом или сферой, его даже с бревном сравнить сложно. В худшем случае дерево, с двумя ветками, расставленными в стороны, но как определить столкновение камеры, которую мы решили представлять как точку с деревом. Как же тогда определить столкновение с деревом/человеком?

Самый точный способ — произвести проверку столкновения с каждым треугольником, из которого состоит сетка персонажа. Это действительно тяжелый труд для процессора, я бы сказал, стахановский труд. Благо нам такая точность не нужна и в большинстве случаев мы можем упростить расчет.

Самый грубый обман — заключить всего человека в бокс и воспринимать его как таковой. Посмотрите на рис. 5.1, где показана девушка с мечом, заключенная в бокс. Преимущество такого метода — это то, что достаточно проверить на столкновение только простую фигуру, что не так уж и сложно, и такую проверку процессор произведет очень быстро. Таким же способом можно заключить персонаж в сферу, если он напоминает больше округлые формы (например, сферический космический корабль).

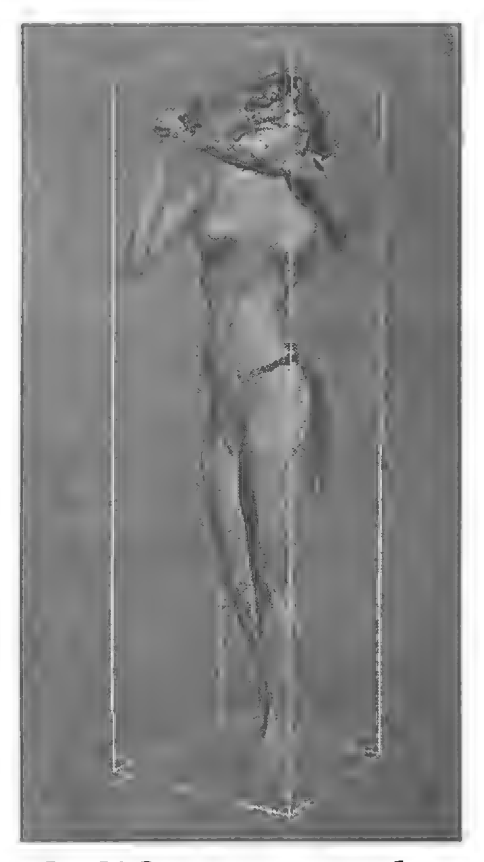


Рис. 5.1. Девушка, заключенная в бокс

Да, это очень грубый способ проверки столкновений, но он очень быстрый. В сочетании с более точными проверками, он позволит решить проблему производительности. Где можно применять такой способ? В первую очередь, для проверки столкновений персонажей со стенами виртуального мира. Да, такая проверка приближенная, и получится, что персонажи/враги не будут подходить к стенам ближе, чем на вытянутую руку, но пользователь этого не заметит. Какая разница, где гуляют наши враги — главное, чтобы они пере-

мещались, а то, что враг не может слишком близко проходить мимо предметов виртуального мира, никто не заметит.

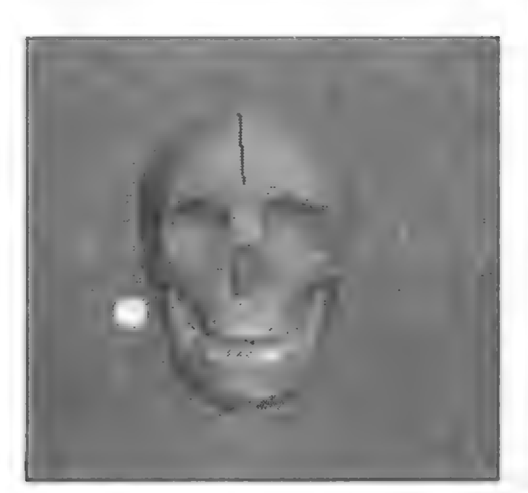
Заключение персонажей в боксы или сферы можно использовать и как базовый метод для проверки на попадания из орудий, т. е. столкновения с пулями. Я бы сказал, что как базовый он может использоваться всегда. Почему базовый? Да потому что игрок просто не простит, если луч выстрела пролетит явно между ногами или рядом с ногой (внутри описывающего персонажа бокса, но явно не пересечет сетку Mesh), а монстр из-за этого умрет. Такие вещи слишком заметны, поэтому нужно быть более аккуратным и проверку производить в два этапа:

1. Произвести проверку попадания пули внутрь куба, описывающего персонаж. Если пуля не попала, то персонаж должен жить и нет смысла на уровне более мелких деталей проверять столкновения. Таким образом, будет быстро отбраковано большинство объектов игры.
2. Если пуля находится в области прямоугольника или сферы, описывающего персонажа/врага, то необходимо потратить время и проверить столкновение на уровне более мелких простых фигур или треугольников.

Таким образом, с помощью быстрой проверки на столкновения сфер и боксов мы быстро находим необходимый объект, с которым произошло пересечение, и только если произошло пересечение описывающего бокса/сферы, используем более сложные и тяжелые для проверки алгоритмы.

В конце описания второго пункта я затронул тему уровней и сказал "более мелких простых фигур". Что это значит? Допустим, что у нас есть череп. Его можно заключить в сферу, потому что голова напоминает именно эту фигуру. Но это не совсем корректно, ведь верх черепа достаточно объемный, а скулы более узкие и получится, что если пуля пролетит внутри сферы, но рядом со щекой, на уровне больших объектов (сферы) мы решим, что произошло столкновение, которого не было. На рис. 5.2 вы можете увидеть такую ситуацию, где слева от скулы внутри описывающего круга в виде белого пятнышка показано попадание снаряда.

Как решить эту проблему? Несмотря на простоту формы черепа, после проверки столкновения с описывающей сферой можно проверять на уровне треугольников сетки. Но для описания сферических объектов необходимо слишком большое количество треугольников. Чем более гладкая поверхность черепа, тем больше треугольников и больше понадобится расчетов. Проблему решает разбиение сложной фигуры на несколько более простых. Например, череп можно разбить на три части и описать сферами по отдельности. Например, на рис. 5.3 череп разделен на три части — вокруг каждой из них показаны боксы.



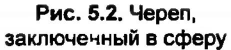


Рис. 5.2. Череп, заключечный в сферу



Рис. 5.3. Череп описан несколькими фигурвми

Теперь описывающие боксы прилегают к сетке достаточно близко и вероятность попадания снаряда внутрь такого бокса, но не попадания его в сетку становится меньше. А если и произойдет такая ситуация, то за взрывом от снаряда пользователь не заметит, что столкновения реально не произошло и поверит тому, что произошло попадание в сетку черепа.

В данном случае для описания фигуры можно использовать и сферы, а можно и то и другое одновременно (верх черепа в виде сферы, а скулы в виде бокса).

5.3. Пример реализации коллизий

Давайте для примера наделим наш мир одной стенкой и персонажем. Комната нам не понадобится, чтобы вы могли двигаться в любых направлениях и обходить виртуальную стену с любой стороны. Теперь добавим к классу работы с людьми метод `getMeshBoundBox`, который будет рассчитывать бокс, описывающий сетку:

```
void CDXGObject::getMeshBoundBox()
{
 BYTE* pVertices;

 // Блокировка поверхности
 if (FAILED(pObjMesh->LockVertexBuffer(D3DLOCK_READONLY,
          (LPVOID*)&pVertices)))
   return;
```

```
 // Рассчитываем бокс, описывающий фигуру
 D3DXComputeBoundingBox((D3DXVECTOR3*)pVertices,
        pObjMesh->GetNumVertices(),
        D3DXGetFVFVertexSize(pObjMesh->GetFVF()),
        &vecBoundingBoxMin, &vecBoundingBoxMax);

 // Разблокируем поверхность
 pObjMesh->UnlockVertexBuffer();
}
```

Вызов getMeshBoundBox необходимо добавить в конец методов загрузки сетки, т. е. в LoadMeshFromFile и LoadSkinMeshFromFile.

Теперь посмотрим, что делает метод для определения описывающего сетку бокса. Нам ничего не нужно самостоятельно рассчитывать, потому что эту задачу может выполнить функция D3DXComputeBoundingBox, которая в общем виде выглядит следующим образом:

```
HRESULT D3DXComputeBoundingBox(
 PVOID pPointsFVF,
 DWORD NumVertices,
 DWORD FVF,
 D3DXVECTOR3* pMin,
 D3DXVECTOR3* pMax
);
```

Метод получает пять параметров:

- pPointsFVF — указатель на буфер, содержащий вершины сетки, вокруг которой необходимо рассчитать бокс;
- NumVertices — количество вершин;
- FVF — формат вершин, который легко определить с помощью метода GetFVF (pObjMesh->GetFVF());
- pMin — переменная-вектор, в которой после выполнения функции будет находиться координата минимальной нижней вершины куба, описывающего сетку;
- pMax — переменная-вектор, в которой после выполнения функции будет находиться координата максимальной верхней вершины куба, описывающего сетку.

Для того чтобы получить указатель на буфер вершин, в самом начале метода getMeshBoundBox, мы блокируем поверхность сетки для получения указателя на буфер.

Последние два параметра определяют две вершины, через которые вы можете построить бокс, описывающий сетку. На рис. 5.4 показаны вершины, а также

как определить координаты. Как видите, это не так уж и сложно. Поскольку двух вершин достаточно, то именно их и возвращает функция.

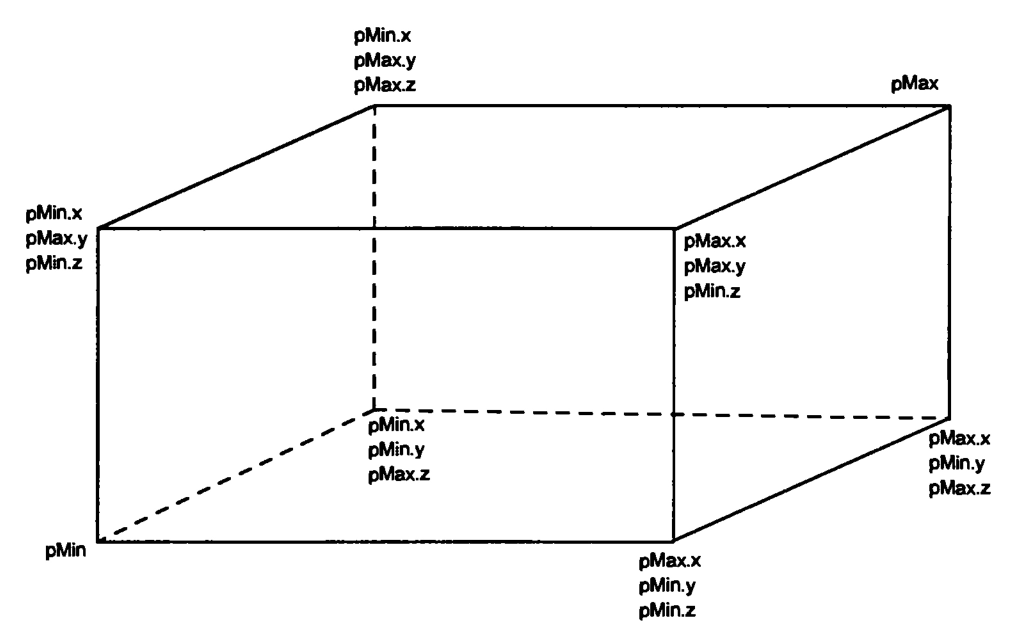


Рис. 5.4. Череп описан несколькими фигурами

Теперь посмотрим, как будет выглядеть функция движения камеры. Именно в этот момент мы должны проверить — произошло столкновение или нет. Если столкновение произошло, то необходимо отбросить камеру от препятствия. Зачем? Попробуйте кинуть какой-либо предмет на стену, и он отскочит назад. Конечно же, мы не будем рассчитывать отскок с учетом физических законов, а просто отбросим камеру немного назад.

Итак, в листинге 5.1 показана функция `MoveCamera` с учетом проверки столкновений.

Листинг 5.1. Функция MoveCamera

```
void CGraphEngine::MoveCamera(float fOffset, D3DXVECTOR3 *vecModifier)
{
 D3DXMATRIX tView, tMat;
 D3DXVECTOR3 vect;

 // Двигаем камеру
 D3DXVec3Subtract(&vect, vecModifier, &pos);
```

```
vect *= fOffset;
D3DXMatrixTranslation(&tView, vect.x, vect.y, vect.z);

// Сохраняем движение во временной матрице tMat
tMat = mvMat;
D3DXMatrixMultiply((D3DXMATRIX*)&tMat, (D3DXMATRIX*)&tMat,
    (D3DXMATRIX*)&tView);

// Цикл проверки столкновений
int i=0;
while(i<11)
{
 D3DXVECTOR3 BoundingBoxMin, BoundingBoxMax;

 // Трансформируем бокс с учетом матрицы положения объекта
 D3DXVec3TransformCoord(&BoundingBoxMin, &objects[i]->vecBoundingBoxMin,
            objects[i]->GetWorldPos());
 D3DXVec3TransformCoord(&BoundingBoxMax, &objects[i]->vecBoundingBoxMax,
            objects[i]->GetWorldPos());

 // Проверяем столкновение
 if (tMat._41 > BoundingBoxMin.x &&
     tMat._41 < BoundingBoxMax.x &&
     tMat._42 > BoundingBoxMin.y &&
     tMat._42 < BoundingBoxMax.y &&
     tMat._43 > BoundingBoxMin.: &&
     tMat._43 < BoundingBoxMax. )
 {
  // Столкновение произошло, по тому сдвигаем объект
  // в противоположную сторону
  D3DXVec3Subtract(&vect, vecMoc.fier, &pos);
  if (fOffset<0)
   vect *= 0.5;
  else
   vect *= -0.5;

  D3DXMatrixTranslation(&tView, vect.x, vect.y, vect.z);
  tMat = mvMat;
  D3DXMatrixMultiply((D3DXMATRIX*)&tMat, (D3DXMATRIX*)&tMat,
          (D3DXMATRIX*)&tView);
  mvMat = tMat;
  return;
 }
```

```
  i+=10;
 }

 // Столкновения нет, копируем временную матрицу в mvMat
 mvMat = tMat;
}
```

В самом начале мы перемещаем матрицу положения с учетом выбранного направления движения. Тут ничего нового нет, поэтому сразу перескочим на цикл проверки столкновений. В данном примере он стал немного странным:

```
int i=0;
while(i<11)
{
  // Проверка столкновений
  i+=10;
}
```

Благодаря такому циклу будут проверены нулевой и десятый элементы массива. Это нормально, просто для этого примера модель человека я поместил в нулевой, а стену в десятый элемент массива.

Внутри цикла мы должны сначала учесть возможные перемещения объекта в мире и переместить туда же наш бокс, описывающий объект. После этого происходит проверка, и если точка камеры внутри бокса, то отбрасываем объект.

Попробуйте запустить пример и пробежаться по миру. Самое интересное, что сквозь стену вы проскочить не сможете, а сквозь человека — без проблем. Почему такая несправедливость, ведь алгоритм одинаковый?

Проблема кроется в том, что человека мы поворачивали, и после трансформации бокса, описывающего персонаж, максимальная точка может оказаться меньше минимальной, поэтому проверка на столкновение и не проходит. В нашем случае координата Z поменялась местами, поэтому добавьте следующую проверку перед тем, как проверять на столкновение:

```
if (BoundingBoxMin.z > BoundingBoxMax.z)
{
 double mx = BoundingBoxMax.z;
 BoundingBoxMax.z = BoundingBoxMin.z;
 BoundingBoxMin.z = mx;
}
```

В данном случае, если координата Z нижней точки больше координаты Z большей точки, то они меняются местами. В реальной программе вы должны

добавить эту проверку для всех трех составляющих, т. е. еще и для оси X и для Y.

Вот теперь, если запустить пример, то вы не сможете проскочить ни сквозь стену, ни сквозь человека. Вот таким простым способом мы реализовали проверку столкновения на уровне боксов. Попробуйте сделать то же самое, если описывать объекты в виде сферы. Чтобы определить сферу, которая описывает Mesh-объект, можно использовать DX-функцию D3DXComputeBoundingBox, которая выглядит следующим образом:

```
HRESULT D3DXComputeBoundingSphere(
   LPD3DXVECTOR3 pFirstPosition,
   DWORD NumVertices,
    DWORD dwStride,
    D3DXVECTOR3 *pCenter,
    FLOAT *pRadius
);
```

Первые три параметра у этой функции точно такие же, как и у D3DXComputeBoundingBox, а последние два это:

- pCenter — вектор, определяющий центр сферы;
- pRadius — радиус сферы.

Чтобы использовать и сферы и боксы, можно добавить в класс описания объекта что-либо для определения, что именно используется для проверки столкновения.

Дабы сделать движок более универсальным, можно перенести проверку столкновения в сам класс управления объектом.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter4\Collision.

5.4. Ниже плинтуса

Давайте опустимся "ниже плинтуса" и рассмотрим наш объект на более низком уровне. Почему-то на сайте **gemaedev.com** очень часто можно встретить вопросы о том, как можно определить буфер вершин и индексов, как по этим данных определить нудный треугольник. Я понимаю, что это может понадобиться для точного определения столкновения в тех сеточных объектах, где треугольников не много, или где сетка сложная для разбиения на простые фигуры, или где нужна хирургическая точность.

Итак, нам понадобятся следующие функции:

```
void CorrectMeshFormat();
void LoadTriangles();
void GetFace(DWORD dwFace, TRIANGLE &triangle);
```

Именно их мы будем рассматривать в этом разделе. Начнем с первой функции. Ее задача подкорректировать формат вершин в сетке. Дело в том, что вершина

```
void CDXGObject::CorrectMeshFormat()
{
 // Описание формата вершины
 D3DVERTEXELEMENT9 decl[]=
 {
  {0, 0, D3DDECLTYPE_FLOAT3,
       D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_POSITION, 0},
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
       D3DDECLUSAGE_NORMAL, 0},
  {0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
       D3DDECLUSAGE_TEXCOORD, 0},
  D3DDECL_END()
 };

 // Клонируем сетку, чтобы был определенный формат вершины
 pObjMesh->CloneMesh(0, (const D3DVERTEXELEMENT9 *)&decl,
     pDevice, &pObjMesh);

 LoadTriangles();
}
```

Дело в том, что для прямого доступа к координатам вершин мы должны знать их формат. Если в игре формат вершин будет разным, то у нас возникнут проблемы — код придется писать универсальным для каждого формата, а это потеря в производительности. Лучше на этапе загрузки сетки подкорректировать ее формат с помощью клонирования, что и происходит в этом методе.

В самом начале задаем формат вершины, который будет описывать позицию вершины, нормаль и текстурную координату. После этого клонируем сетку с помощью метода `CloneMesh` и с учетом описанной вершины. Результат клонирования записываем в ту же переменную, т. е. перезаписываем сетку.

После того как формат сетки подкорректирован, вызывается функция `LoadTriangles`, которая загружает информацию о треугольниках. Код этой функции вы можете увидеть в листинге 5.2.

Листинг 5.2. Функция загрузки треугольников из сетки

```
void CDXGObject::LoadTriangles()
{
  MeshVertex *pVerts = NULL;
  WORD *pInd = NULL;

  // Определяем количество вершин и выделяем память
  // для сохранения информации о них
  m_dwNumTriangles = pObjMesh->GetNumFaces();
  m_pTriangles= new TRIANGLE[m_dwNumTriangles];

  // Блокируем буфер вершин сетки
  pObjMesh->LockVertexBuffer( D3DLOCK_READONLY,
               reinterpret_cast<VOID**>(&pVerts) );

  // Блокируем буфер индексов
  pObjMesh->LockIndexBuffer( D3DLOCK_READONLY,
               reinterpret_cast<VOID**>(&pInd) );

  // Цикл просмотра всех треугольников
  for( DWORD dwFace = 0; dwFace < m_dwNumTriangles; dwFace++ )
  {
    // Сохраняем очередную вершину
    m_pTriangles[dwFace].v0 = pVerts[pInd[dwFace*3 + 0]].vPosition;
    m_pTriangles[dwFace].v1 = pVerts[pInd[dwFace*3 + 1]].vPosition;
    m_pTriangles[dwFace].v2 = pVerts[pInd[dwFace*3 + 2]].vPosition;

    D3DXVECTOR3 e0 = m_pTriangles[dwFace].v1 - m_pTriangles[dwFace].v0;
    D3DXVECTOR3 e1 = m_pTriangles[dwFace].v2 - m_pTriangles[dwFace].v0;
    D3DXVec3Cross( &(m_pTriangles[dwFace].N), &e0, &e1 );
    D3DXVec3Normalize(&(m_pTriangles[dwFace].N),
        &(m_pTriangles[dwFace].N));

   m_pTriangles[dwFace].d = -D3DXVec3Dot(&(m_pTriangles[dwFace].N),
        &(m_pTriangles[dwFace].v0) );
  }

  // Разблокируем поверхность
  pObjMesh->UnlockVertexBuffer();
  pObjMesh->UnlockIndexBuffer();
}
```

Тут необходимо еще сказать о формате.

```
struct TRIANGLE
{
 D3DXVECTOR3 v0; // Вершина 0
 D3DXVECTOR3 v1; // Вершина 1
 D3DXVECTOR3 v2; // Вершина 2
 D3DXVECTOR3 N;  // Нормаль
 float d;        // Расстояние от центра
};
```

Помимо этого, в наших расчетах участвуют две переменные:

- `TRIANGLE *m_pTriangles;`
- `DWORD m_dwNumTriangles;`

Первая — это массив треугольников, а вторая — это количество треугольников в нашей сцене.

Теперь по комментариям вы можете без проблем разобраться с тем, как мы получаем треугольники сцены. Но один комментарий я все же дам. Для получения информации о вершинах мы блокируем буфер вершин и буфер индексов. На сколько мне известно, сетки всегда используют индексы. Я пока не видел сеток, которые этого не делают.

Теперь мы подошли к тому, к чему и готовились — к функции `GetFace`, которая должна возвращать указанную грань. Функция может выглядеть следующим образом:

```
TRIANGLE *CDXGObject::GetFace(DWORD dwFace)
{
 triangle.v0 = m_pTriangles[dwFace].v0;
 triangle.v1 = m_pTriangles[dwFace].v1;
 triangle.v2 = m_pTriangles[dwFace].v2;
 triangle.N  = m_pTriangles[dwFace].N;
 triangle.d  = m_pTriangles[dwFace].d;
}
```

В первом параметре функция получает номер грани, а итог — структура `TRIANGLE`, через которую будет возвращен результат. Так как эта функция может вызываться для всех граней, которые есть в сетке (а их много), то будем экономить на каждой операции, и никаких проверок не будет. Можно даже сделать переменную `m_pTriangles` открытой (`public`) для прямого доступа к поверхности из внешних классов, чтобы не вызывать лишних функций, уж слишком накладно это делать.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter5\MeshTriangle.

5.5. Напутствие

Для упрощения жизни, в наших примерах камера выглядела как точка, но ведь камера — это тоже персонаж игры, который имеет объем, а не точку. Так как в нашем случае этого персонажа не видно, можно воспринимать камеру как сферу. Как мы убедились, расчеты столкновения сферой не так сложны, главное определить расстояние до центра сферы и сравнить его с радиусом.

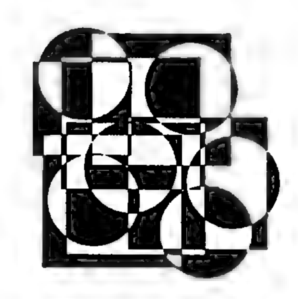
Проверка пересечений даже в самом простом варианте может отнять слишком много ресурсов. Если на одном уровне вашей игры находятся 1000 игровых объектов (вполне реальная цифра), то после каждого перемещения камеры приходится выполнять цикл из 1000 шагов. Даже если каждый шаг будет выполнять всего 8 операций сравнений, в результате мы получаем слишком большие накладные расходы, которые лучше потратить на что-то более интересное.

Как можно ускорить процесс проверки на столкновения? Конечно же, разбить уровень на подуровни. Зачем проверять на столкновения объекты, которые находятся в другой комнате, пусть и того же игрового уровня! Если ограничиться небольшими окнами, то проверка будет простой — всего четыре стены, пару дверей и несколько врагов в этой комнате. Получается, что даже в достаточно сложной комнате может быть не более 100 объектов.

Если сцена происходит на улице и вокруг вас дома, не обращайте внимания на то, что в эти дома можно входить и бродить внутри. Воспринимайте весь дом как один большой бокс. Только после того, как игрок открыл дверь в дом и вошел в комнату, можно воспринимать эту комнату как отдельный участок игры.

Разбиение на комнаты позволяет ускорить не только отсечение, но и отображение. В функции `RenderScene` нужно отображать только те объекты игры, которые находятся в данном игровом участке.

ГЛАВА 6



Вершинная анимация

В *главе 3* мы говорили о скелетной анимации и убедились, как просто она позволяет создать анимацию движения персонажей. Таким способом можно анимировать все, где есть кости. Проблема в том, что кости есть далеко не везде. Например, если танк еще можно сделать на основе скелета, башню и платформу объединить через кости, то вот анимацию губ и глаз сделать через скелет просто невозможно. Я даже представить себе не могу эти органы тела из костей и как эти кости туда засовывать ☺.

Догадайтесь, с чем нам придется работать при вершинной анимации? Вы не поверите, но это будут вершины. Именно их мы будем двигать для создания эффекта анимации. В предыдущей главе, когда мы рассматривали столкновения, в *разд. 5.4* мы узнали, как можно получить вершины, а сейчас нам предстоит анимировать их. Именно поэтому вершинная анимация рассматривается как раз сейчас, когда мы научились работать с вершинами.

Если вы работали в 3D графических программах, то наверно уже не раз редактировали объекты на уровне вершин. В этом разделе мы познакомимся, как с помощью редактирования вершин можно сделать анимацию.

Когда еще не было скелетной анимации, все игровые движки использовали именно вершинную анимацию, двигая вершины сеток всех объектов сцены. Да, это сложнее и в большинстве случаев отнимает много времени, особенно у аниматора, которому приходится создавать ключевые кадры. Но сложность это не единственная проблема — каждый ключ это сетка, которая отнимает больше места в памяти, чем ключи скелетной анимации.

6.1. Теория

Одним из вариантов вершинной анимации является *морфинг*. Все мы видели подобные эффекты уже не раз, как в компьютерных играх, так и в кино. Не

знаю, где именно появился морфинг, но этот эффект получил достаточно широкое распространение и позволяет решить множество задач.

Итак, давайте посмотрим, на чем основан и как реализуется морфинг. Для примера возьмем превращение куба в сферу. Что-то подобное есть среди хранителей экрана под именем "Метаморфозы" (рис. 6.1). В этом хранителе происходит плавное превращение из куба в сферу, потом получается что-то на подобии звезды и т. д.



Рис. 6.1. Хранитель экрана "Метаморфозы"

На компакт-диске в каталоге \Samples\Meta вы можете увидеть пример метаморфоз, который я рассматривал в книге [3]. Для анимации используются шейдер, код которого можно увидеть в листинге 6.1, потому что это проще. Вершинный шейдер получает на входе координаты вершины, а наша задача только подкорректировать позицию с учетом определенного коэффициента или прошедшего времени.

Листинг 6.1. Шейдер, реализующий анимацию морфингом

```
float4x4 mat;
float4x4 worldmat;
```

```
float ViewAngle;
float Min = -3.0f;
float Max = 3.0f;
float4 lightDir = {0.0, 4.0, -5.0, 1};

// Выходные данные
struct VS_OUTPUT
{
   float4 Pos:     POSITION;  // Позиция
   float3 normal:  TEXCOORD0; // Нормаль
   float3 viewvec: TEXCOORD1; // Вектор просмотра
   float4 basecol: TEXCOORD2; // Базовый цвет
};

// Вершинный шейдер
VS_OUTPUT Vexel_Sh(float4 Pos: POSITION,
  float3 normal: NORMAL)
{
   VS_OUTPUT Out;

   // Нормализуем позицию
   float3 spherePos = normalize(Pos.xyz);

   // Расчет позиции
   float t = frac(ViewAngle);
   t = smoothstep(0, 0.5, t) - smoothstep(0.5, 1, t);

   float lrp = Min + (Max - Min) * t;

   Pos.xyz = lerp(spherePos, Pos, lrp);
   normal = lerp(spherePos, normal, lrp);

   // Изменяем цвет
   Out.basecol = 0.5 + 0.5 * Pos;

   // Корректируем положение с учетом матрицы положения
   Out.Pos = mul(Pos, mat);
   Out.normal = mul(normal, worldmat);
   Out.viewvec = mul(Pos, worldmat);

   return Out;
}
```

```
// Пиксельный шейдер
float4 Pixel_Sh(float3 normal: TEXCOORD0, float3 viewvec: TEXCOORD1,
  float4 basecol: TEXCOORD2) : COLOR
{
    // Расчет освещения пиксела
    float3 normalized = normalize(normal);
    float3 lightn = normalize(lightDir);
    float4 diffuse = saturate(dot(lightDir, normalized));

    float3 reflection = normalize(2 * diffuse * normalized - lightn);
    float3 eyen = normalize(viewvec);
    float4 specular = pow(saturate(dot(reflection, eyen)), 16);

    return basecol * diffuse + specular;
}

// Техника отображения
technique PixelLight
{
    pass P0
    {
        VertexShader = compile vs_2_0 Vexel_Sh();
        PixelShader  = compile ps_2_0 Pixel_Sh();
    }
}
```

Использование шейдера для морфинга удобно, но не всегда возможно. В связи с этим, нам придется программно получать данные о вершинах сетки и управлять ими. Да, это требует больше ресурсов и необходима блокировка поверхностей, но этот метод универсальный и будет работать на всех видеокартах, ведь шейдеры есть не везде.

Тут необходимо заметить одно правило — *во время анимации количество вершин в первичном и конечном объекте должно быть одинаковым*. Если количество будет отличаться, то вы не сможете определить, когда, как и где нужно добавить или удалить определенную вершину. Результирующая сцена в этом случае может оказаться очень не красивой, я бы сказал — страшнее Омена.

Должна совпадать и последовательность вершин. Если будет нарушена последовательность расположения вершин, то результат может оказаться неожиданным.

Чтобы во время вершинной анимации не было проблем, ключевые кадры лучше всего создавать на основе сетки, которую необходимо анимировать.

Эти кадры нужно создавать простым перемещением вершин и сохранением результирующей сетки в файл. Получается, что у вас будет несколько одинаковых сеток, где положение некоторых или всех вершин может быть изменено. Ваша задача, в зависимости от прошедшего времени рассчитать промежуточные положения точек, как мы это делали при скелетной анимации.

Для практического примера я нашел модель 3D-рыцаря на лошади (рис. 6.2). Чтобы создать два ключа анимации, сначала я сохранил исходное состояние сцены в файле monster.x. После этого я приподнял ноги лошади, редактируя всадника на уровне вершин с помощью 3D-редактора, и сохранил результат в файле monsterkey.x.

Теперь у меня есть два файла, содержащих всадника в двух состояниях. Хочу заметить, что модель лучше редактировать в одном состоянии. Например, если сетка создана как Mesh, то не меняйте режим редактирования на полигоны (Poly). Во время смены режима последовательность вершин нарушается, и результат будет ужасным. На экране может появиться что угодно. Я предпочитаю редактировать сцену как Mesh на уровне вершин (Editable Mesh). Это замечание касается 3D Studio Max, в других 3D-редакторах я не пробовал редактировать объекты. Возможно, данный эффект проявляется из-за того, что при смене режима происходит изменения порядка следования вершин, а возможно, из-за моего плохого знания этого пакета. Да, я знаю 3D Studio только на уровне выполнения основных операций.

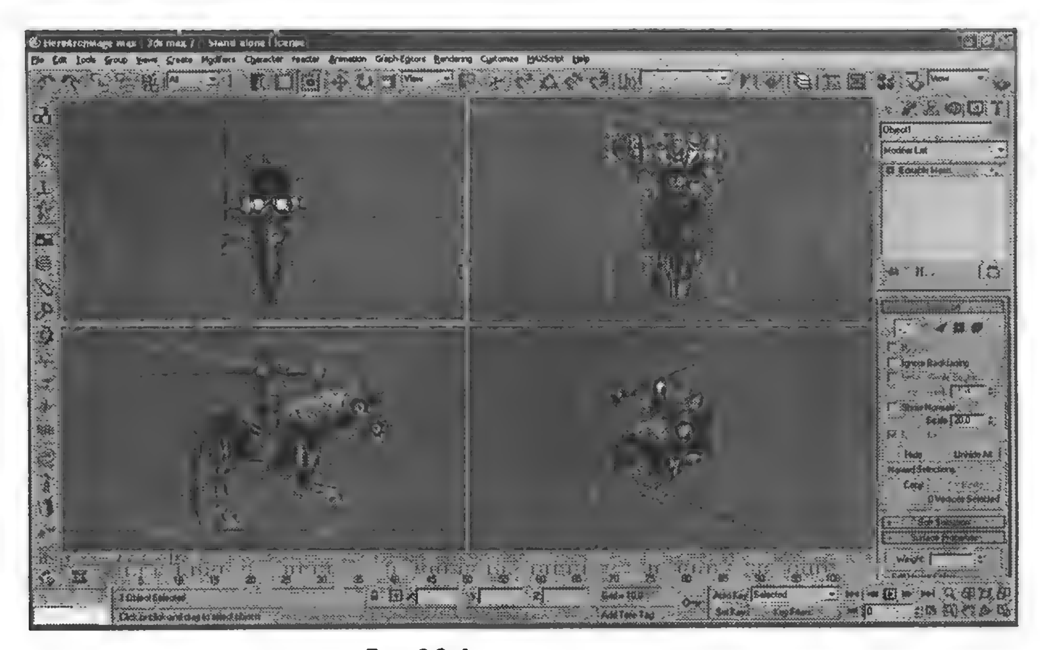


Рис. 6.2. Анимируемая модель

6.2. Загрузка ключа

Для начала нашему движку необходимо добавить возможность загружать ключевые сетки. Для этого в классе управления объектами `CDXGObject` добавим следующие две переменные:

- `ID3DXMesh *pMeshKey1;`
- `ID3DXMesh *pMeshToDraw.`

Первая переменная будет использоваться для хранения указателя на сетку ключевого кадра. У нас их всего два — исходное состояние, которое будет храниться в переменной `pObjMesh`, где мы хранили сетку объекта и раньше `pMeshKey1`, для хранения конечного состояния.

Переменную `pMeshToDraw` будем использовать для хранения сетки, содержимое которой нужно отобразить. У нас есть только крайние положения сетки, а промежуточные приходится рассчитывать и сохранять в переменной `pMeshToDraw` для последующего отображения.

Теперь добавим нашему классу метод `LoadKeyFromFile`, который будет загружать ключевую сетку и выполнять все необходимые действия по инициализации. Этот метод будет выглядеть, как показано в листинге 6.2.

Листинг 6.2. Метод загрузки ключевой сетки

```
void CDXGObject::LoadKeyFromFile(char* filename, char* texturename)
{
 LPDIRECT3DTEXTURE9 *pTextures;
 D3DMATERIAL9 *pMaterials;

 // Загрузка сетки
 int dwNum = LoadMesh(filename, pDevice,
   &pMeshKey1, &pTextures, texturename, &pMaterials);

 // Формат вершины
 D3DVERTEXELEMENT9 decl[]=
 {
  {0, 0,  D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_POSITION, 0},
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_NORMAL, 0},
  {0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_TEXCOORD, 0},
  D3DDECL_END()
 };
```

```
 // Клонируем сетку, чтобы получить одинаковый формат вершины
 pMeshKey1->CloneMesh(0, (const D3DVERTEXELEMENT9 *)&decl,
          pDevice, &pMeshKey1);
 pMeshKey1->CloneMesh(0, (const D3DVERTEXELEMENT9 *)&decl,
          pDevice, &pMeshToDraw);
}
```

Для загрузки сетки используем уже знакомую нам функцию LoadMesh. После этого объявляем описание вершины, чтобы клонировать сетку и гарантировать, что формат вершины будет содержать позицию, нормаль и координату текстуры. Клонирование производим дважды в переменную pMeshKey1, т. е. в саму себя и в переменную pMeshToDraw. Второе клонирование необходимо для того, чтобы проинициализировать переменную pMeshToDraw.

Теперь, после загрузки сетки добавляем строку загрузки ключевого кадра:

```
objects[index]->LoadKeyFromFile("Media\\monsterkey.x",
     "Media\\texture.bmp");
```

6.3. Отображение сетки

У нас уже загружены оба ключа, и теперь можно переходить к отображению. Для этого придется модифицировать метод Render у класса управления объектом. Новый вариант можно увидеть в листинге 6.3.

Листинг 6.3. Отображение сцены

```
bool CDXGObject::Render(DWORD ef_index, int light)
{
 // Здесь опущена настройка отображения
 ...

 if (pMeshKey1)
 {
  double tCurrentTime = GetTickCount();

  // Время анимации
  if ((tCurrentTime-tStartAnimTime)>4000)
   tStartAnimTime = GetTickCount();
  double t = (tCurrentTime - tStartAnimTime) / 4000;

  // Промежуточные переменные
  MeshVertex *pObj;
  MeshVertex *pKey1;
  MeshVertex *pDraw;
```

```
  // Блокируем поверхности сеток
  pObjMesh->LockVertexBuffer(D3DLOCK_READONLY, (void**)(&pObj));
  pMeshKey1->LockVertexBuffer(D3DLOCK_READONLY, (void**)(&pKey1));
  pMeshToDraw->LockVertexBuffer(0, (void**)(&pDraw));

  // Цикл расчета сетки на текущий момент времени
  for (DWORD dw = 0; dw < pObjMesh->GetNumVertices(); dw++)
   pDraw[dw].vPosition = pKey1[dw].vPosition * t +
        pObj[dw].vPosition * (1 - t);

  // Разблокируем поверхность
  pMeshToDraw->UnlockVertexBuffer();
  pMeshKey1->UnlockVertexBuffer();
  pObjMesh->UnlockVertexBuffer();
 }

 // Отображение объекта
 UINT uPass;
 pEffect[ef_index]->Begin(&uPass, NULL);

 // Цикл перебора шагов техники отображения
 for(UINT i = 0; i < uPass; i++)
 {
  // Здесь опущен код отображения
  ...
 }
}
```

Код очень простой, но кратко пробежимся по нему. В самом начале проверяем, если переменная `pMeshKey1` существует, значит для данной сетки есть ключевой кадр. Эта проверка нужна только для данного примера, а в реальных условиях придется придумывать другой метод определения метода анимации для текущего объекта.

После этого заводим три переменные типа `MeshVertex`. Напоминаю, это структура, которая описывает вершину в сетке:

```
struct MeshVertex
{
 D3DXVECTOR3 vPosition;
 D3DXVECTOR3 vNormal;
 D3DXVECTOR2 vTexCoord;
};
```

Давайте посмотрим, для чего нужны эти переменные:

- `pObj` — для хранения указателя на буфер вершин начального состояния;
- `pKey1` — для хранения указателя на буфер вершин ключевого кадра, который содержит сетку конечного положения;

- pDraw — для хранения указателя на буфер вершин сетки промежуточного состояния, которую мы будем отображать.

После этого блокируем буферы вершин всех трех сеток и запускаем цикл перебора всех вершин и расчета их промежуточного положения. Этот цикл завершиться неудачей, если количество вершин не будет совпадать. В нашем случае сетка для обоих кадров одна и та же (в 3D-редакторе я только модифицировал положение некоторых точек), поэтому ошибок не будет.

После цикла не забываем разблокировать поверхности, чтобы не вызвать проблем, и можно считать, что сцена готова.

Запустите пример и убедитесь, что лошадь приподнимает передние копыта.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter6\ VertexAnim.

6.4. Использование шейдера

Как я уже говорил, для морфинга можно использовать и шейдер, в этом случае пример будет работать быстрее. Давайте попробуем посмотреть, как шейдер может использоваться для анимации с использованием ключевых сеток. Мы рассмотрим необходимую теорию, а реализацию оставим в качестве домашнего задания, так что на компакт диске не будет нужного примера, я его просто не писал.

Итак, чтобы шейдер мог рассчитать положение вершины, ему необходимо знать начальное и конечное положение. Как передать эти два значения? Для начала необходимо описать структуру входных данных следующим образом:

```
struct VS_INPUT
{
   // Данные для точки 0
   float3 Pos:      POSITION0;
   float3 Normal:   NORMAL0;
   float2 TexCoord: TEXCOORD0;

   // Данные для точки 1
   float3 Pos1:      POSITION1;
   float3 Normal1:   NORMAL1;
   float2 TexCoord1: TEXCOORD1;
};
```

Уже по объявлению можно догадаться, что необходимо использовать два потока. В первый поток помещаем сетку с исходным состоянием, а во второй кадр с конечным состоянием.

В шейдер необходимо еще добавить переменную mTime, которая будет определять прошедшее время:

```
float mTime;
```

Теперь функция вершинного шейдера будет выглядеть следующим образом:

```
VS_OUTPUT Vexel_ShObj(VS_INPUT In)
{
 VS_OUTPUT Out;

 Out.Pos = float4(In.Pos1, 1)*mTime+float4(In.Pos, 1)*(1-mTime);
 Out.Pos = mul(Out.Pos, world_view_proj_matrix);

 Out.TexCoord = In.TexCoord;
 Out.Normal = In.Normal1;

 return Out;
}
```

Теперь переходим к рассмотрению кода. Для начала и здесь нужно подкорректировать формат вершины:

```
D3DVERTEXELEMENT9 decl[]=
{
 // Поток 1
 {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_POSITION, 0},
 {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_NORMAL, 0},
 {0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_TEXCOORD, 0},

 // Поток 2
 {1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_POSITION, 1},
 {1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_NORMAL, 1},
 {1, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_TEXCOORD, 1},

 D3DDECL_END()
};
```

Первые три записи определяют формат вершины в первом потоке, а следующие три записи — это формат вершины во втором потоке. Регистрируем этот формат как всегда:

```
pDevice->CreateVertexDeclaration(decl, &VertDecl);
pDevice->SetVertexDeclaration(VertDecl);
```

В листинге 6.4 я набросал код, который может использоваться для визуализации. Подробные комментарии помогут вам разобраться с этим примером.

Листинг 6.4. Пример реализации морфинга через шейдер

```
// Расчет коэффициента времени анимации
double tCurrentTime = GetTickCount();

if ((tCurrentTime-tStartAnimTime)>4000)
 tStartAnimTime = GetTickCount();

float t = (tCurrentTime - tStartAnimTime) / 4000;

// Устанавливаем переменную mTime (прошедшее время)
if (pEffect[ef_index]->IsParameterUsed("mTime",
pTechniqueName[ef_index]))
 pEffect[ef_index]->SetValue("mTime", &t, D3DX_DEFAULT);

// Начинаем шейдер
UINT uPass;
pEffect[ef_index]->Begin(&uPass, NULL);

// Запускаем цикл отображения через шейдер
for (UINT i = 0; i < uPass; i++)
{
 pEffect[ef_index]->Pass(i);

 // Устанавливаем текстуру, если она есть
 if (pMeshTextures[0])
 {
  pDevice->SetTexture(0, pMeshTextures[0]);
  if (pEffect[ef_index]->IsParameterUsed("tText",
          pTechniqueName[ef_index]))
    pEffect[ef_index]->SetValue("tText", &pMeshTextures[0],
          D3DX_DEFAULT);
  }

  // Объявляем переменные для хранения вершинного и индексного буфера
  LPDIRECT3DVERTEXBUFFER9 pVB;
  LPDIRECT3DINDEXBUFFER9 pIB;

  // Переменная для вершинного буфера второго потока
  LPDIRECT3DVERTEXBUFFER9 pVBKey;
```

```
  // Получаем буфер вершин и индексов первой сетки
  pObjMesh->GetVertexBuffer(&pVB);
  pObjMesh->GetIndexBuffer(&pIB);

  // Получаем количество вершин и поверхностей
  DWORD numVertices = pObjMesh->GetNumVertices();
  DWORD numFaces = pObjMesh->GetNumFaces();

  // Установить буфер первого кадра в нулевой поток
  pDevice->SetStreamSource(0, pVB, 0,
         D3DXGetFVFVertexSize(pObjMesh->GetFVF()));

  // Установить индексы
  pDevice->SetIndices(pIB);

  // Получить буфер вершин второй сетки
  pMeshKey1->GetVertexBuffer(&pVBKey);

  // Установить буфер второй сетки в 1-й поток
  pDevice->SetStreamSource(1, pVBKey, 0,
        D3DXGetFVFVertexSize(pMeshKey1->GetFVF()));

  // Отобразить буфер вершин
  pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
         numVertices, 0, numFaces);

  // Обнулить потоки
  pDevice->SetStreamSource(0, NULL, 0, 0);
  pDevice->SetStreamSource(1, NULL, 0, 0);

  // Уничтожить временные переменные буферов
  if (pVBKey){pVBKey->Release(); pVBKey=NULL;}
  if (pVB){pVB->Release(); pVB=NULL;}
  if (pIB){pIB->Release(); pIB=NULL;}
 }
pEffect[ef_index]->End();
```

Все крутится вокруг вызова метода `SetStreamSource`. В качестве первого потока устанавливаем буфер вершин первого кадра (первый параметр равен нулю):

```
pDevice->SetStreamSource(0, pVB, 0,
         D3DXGetFVFVertexSize(pObjMesh->GetFVF()));
```

В качестве второго потока устанавливаем вершины второй сетки (первый параметр равен единице):

```
pDevice->SetStreamSource(1, pVBKey, 0,
        D3DXGetFVFVertexSize(pMeshKey1->GetFVF()));
```

Шейдер будет получать по одной вершине из каждого потока и рассчитывать положение с учетом прошедшего времени. Для шейдеров, как и для блокировки, количество вершин в первой и последней сетке должно быть одинаковым, иначе сцена будет искаженной.

Как я уже сказал, на компакт-диске нет именно такого примера, потому что здесь показан упрощенный вариант, который хорош для иллюстрации. Зато на диске есть пример, который показывает внедрение данного варианта морфинга в наш простой, но уже достаточно хороший движок. Сразу хочу предупредить, что шейдер для теней я не создавал и его придется подкорректировать.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter6\MorthWithShader.

6.5. Множественность кадров

В данной главе мы рассмотрели пример, в котором происходит движение только между двумя кадрами. А как сделать движение между несколькими кадрами? Рассматривать этот вариант мы не будем, чтобы вы сами попробовали реализовать это решение, но вот о теории поговорим.

Итак, для создания анимации между множествами кадров можно завести дополнительный класс, который будет похож на рассмотренный нами в *главе 3*:

```
class CMorthAnimationSet
{
protected:
 IDirect3DDevice9 *pDevice;

public:
 DWORD EndTime;
 ID3DXMesh *pMeshKey;
 CMorthAnimationSet *NextMorthAnimSet;

 CMorthAnimationSet(IDirect3DDevice9 *pD3DDevice);
 ~CMorthAnimationSet();
 void LoadKeyFromFile(char* filename, char* texturename);
};
```

В данном примере мы объявляем класс CMorthAnimationSet, который будет хранить ключ вершинной анимации. У этого класса всего четыре поля (первое является закрытым от внешнего воздействия):

- pDevice — указатель на устройство Direct3D;
- EndTime — время, до которого текущая анимация должна завершиться;
- pMeshKey — сетка, которая определяет форму объекта в конце анимации;
- NextMorthAnimSet — следующая анимация. Если этот параметр равен нулю, то это последняя анимация.

Помимо этого у нас есть еще конструктор и деструктор, в которых задаем значения по умолчанию. Для загрузки сетки используем метод LoadKeyFromFile, код которого перекочевал сюда из класса CDXGObject, поэтому его мы здесь рассматривать не будем.

В листинге 6.5 показано, как можно загрузить базовую сетку и два ключа анимации.

Листинг 6.5. Загрузка ключей анимации

```
// Загрузка базовой сетки
objects[index]->LoadMeshFromFile("Media\\monster.x",
      "Media\\HEROARCH.bmp");

// Создание и загрузка ключа 1
objects[index]->MorthAnimation = new CMorthAnimationSet(pDevice);
objects[index]->MorthAnimation->LoadKeyFromFile("Media\\monsterkey.x",
    "Media\\HEROARCH.bmp");
objects[index]->MorthAnimation->EndTime = 4000;

// Создание ключа 2
objects[index]->MorthAnimation->NextMorthAnimSet =
     new CMorthAnimationSet(pDevice);

// Сохраняем указатель на созданный ключ и загружаем в него сетку
CMorthAnimationSet *ca=objects[index]->MorthAnimation->NextMorthAnimSet;
ca->LoadKeyFromFile("Media\\monsterkey2.x", "Media\\HEROARCH.bmp");
ca->EndTime = 8000;

// Устанавливаем эффект и технику отображения
objects[index]->SetEffect(pEffect[3], 0);
objects[index]->SetTechnique("PixelLight",  0);
break;
```

В качестве времени анимации для первого ключа выбрано 4000, т. е. от базовой сетки до первого ключа превращение будет происходить в течение 4 с. Второму ключу время анимации указано 8000, а значит, анимация будет длиться от конца времени анимации предыдущего кадра (от 4000 миллисекунд) до 8000, т. е. снова 4 с. Если вы захотите создать еще и третий ключ анимации, то код его создания (и всех последующих ключей) может выглядеть следующим образом:

```
// Создать ключ для следующей анимации
ca->NextMorthAnimSet = new CMorthAnimationSet(pDevice);

// Сохраняем следующую анимацию в качестве текущей
ca = ca->NextMorthAnimSet;

// Загрузить сетку
ca->LoadKeyFromFile("имя файла", "текстура");

// Указать время
ca->EndTime = время;
```

Теперь посмотрим, как может выглядеть код отображения сцены с помощью такого класса. Этот код показан в листинге 6.6. Это часть метода `Render` класса управления объектом.

Листинг 6.6. Отображение вершинной анимации из нескольких ключей

```
bool CDXGObject::Render(DWORD ef_index, int light)
{
 // Здесь идет задание основных параметров
 ...

 // Переменные для анимации
 IDirect3DVertexDeclaration9 *VertDecl;     // Формат вершины
 ID3DXMesh *Key1Mesh = pObjMesh;            // Первая ключевая сетка
 CMorthAnimationSet *ca2 = MorthAnimation; // Текущая анимация

 // Если необходимо использовать анимацию, то ...
 if (MorthAnimation)
 {
  double tCurrentTime = GetTickCount();
  DWORD StartTime = 0;

  // Цикл поиска текущего ключа анимации
  while (ca2)
```

```
  {
   // Если время анимации больше конца анимации текущего ключа, то
   if ((tCurrentTime - tStartAnimTime) > ca2->EndTime)
   {
    // берем следующий ключ
    StartTime = ca2->EndTime;
    Key1Mesh=ca2->pMeshKey;
    ca2=ca2->NextMorthAnimSet;
   }
   else
    break;
  }

  // Если ключ не найден, значит нужно начать с начала
  if (ca2==NULL)
  {
   tStartAnimTime = GetTickCount();
   ca2 = MorthAnimation;
  }

  // Расчет коэффициента
  float t = (tCurrentTime - tStartAnimTime - StartTime) /
      (ca2->EndTime - StartTime);

  // Установить коэффициент
  if (pEffect[ef_index]->IsParameterUsed("mTime",
      pTechniqueName[ef_index]))
    pEffect[ef_index]->SetValue("mTime", &t, D3DX_DEFAULT);

  // Здесь необходимо задать формат вершины для двух потоков данных
  D3DVERTEXELEMENT9 decl[]=
  {
    ...
  }
 }
 else
 {
   // Анимации нет, формат вершин задаем с одним потоком данных
   D3DVERTEXELEMENT9 decl[]=
   {
     ...
   };
   pDevice->CreateVertexDeclaration(decl, &VertDecl);
   pDevice->SetVertexDeclaration(VertDecl);
  }
```

```
// Отображаем объект
UINT uPass;
pEffect[ef_index]->Begin(&uPass, NULL);

for (UINT i = 0; i < uPass; i++)
{
 // Здесь начинаем анимацию и задаем текстуру

 if (MorthAnimation)
 {
  // Получаем данные сетки начала текущей анимации
  Key1Mesh->GetVertexBuffer(&pVB);
  Key1Mesh->GetIndexBuffer(&pIB);

  DWORD numVertices = Key1Mesh->GetNumVertices();
  DWORD numFaces = Key1Mesh->GetNumFaces();

  pDevice->SetStreamSource(0, pVB, 0,
     D3DXGetFVFVertexSize(Key1Mesh->GetFVF()));
  pDevice->SetIndices(pIB);

  // Получаем данные конечной сетки анимации
  ca2->pMeshKey->GetVertexBuffer(&pVBKey);
  pDevice->SetStreamSource(1, pVBKey, 0,
  D3DXGetFVFVertexSize(ca2->pMeshKey->GetFVF()));

  pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
    numVertices, 0, numFaces);

  pDevice->SetStreamSource(1, NULL, 0, 0);
  if (pVBKey){pVBKey->Release(); pVBKey=NULL;}
 }
 else
 {
  // Нет анимации, отображение стандартным способом
 }
 if (pVB){pVB->Release(); pVB=NULL;}
 if (pIB){pIB->Release(); pIB=NULL;}
}
pEffect[ef_index]->End();

return true;
}
```

Самое интересное — это цикл поиска пары ключей, которые должны сейчас использоваться для расчета положения вершин в сетке. Подробные комментарии помогут вам разобраться с тем, как это происходит.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter6\MorthWithShader2.

Рассмотренный в этом разделе код нельзя считать законченным. Это всего лишь набросок, который показывает, как можно сделать анимацию из неопределенного количества ключей. Код получился очень неуклюжим, потому что слишком универсальный и позволяет отображать не только статичные объекты, но и объекты со скелетной или вершинной анимацией. В реальных условиях я бы создал несколько методов `Render`, каждый для определенного типа объекта. Но как я сказал в начале раздела, создание законченного варианта я оставлю на ваше усмотрение.

6.6. Практика использования

Некоторые игры полностью основаны на вершинной анимации 3D-объектов. К преимуществам метода можно отнести возможность анимации совершенно любых объектов, как мягких тел (одежда, губы, веки на глазах), так и твердых тел, таких как ноги. Получается, что мы можем полностью отказаться от использования скелетной анимации. К недостаткам метода можно отнести большую потребность в необходимой памяти. Каждый ключевой кадр требует существенного количества памяти для хранения полноценной сетки объекта.

При использовании обоих методов вы можете возложить создание моделей на профессионала в своей области. Если я не ошибаюсь, то таких людей называют 3D-аниматорами. Для создания кадров вершинной анимации можно использовать в 3D-редакторе те инструменты, что и для скелетной анимации, но в файле не нужно сохранять скелет, необходима результирующая сетка, которая получается после поворотов костей.

Скелетная анимация требует намного меньше памяти для хранения каждого кадра анимации, но при этом таким методом можно анимировать далеко не все. Вы можете получить преимущества от использования и той и другой технологии. Тело персонажа может быть сделано из костей и кожи, а значит, анимироваться по методу скелетной анимации, а вот глаза и рот можно как бы наклеить поверх скелета. Да, это немного неудобно и усложняет код, но зато значительно экономится память, которой много никогда не бывает.

Чтобы проще было жить, с помощью вершин можно анимировать целую голову, а вот все остальное уже с помощью скелета. Так проще, потому что не-

обходимо соединить только два объекта с разным типом анимации (голова и туловище), а не четыре (туловище, два глаза и губы). Соединение можно сделать в районе шеи.

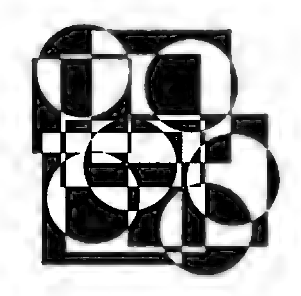
Если на персонаже есть одежда и если она не прилегает к телу, то она должна как-то шевелиться. Такие предметы одежды, как пиджак, галстук, шарф, плащ и т. д., должны колыхаться на ветру, а анимировать одежду с помощью скелета также нереально. Натянуть на сетку скелетной анимации сетку с вершинной анимации — глупо. Такие персонажи лучше полностью делать из вершин.

Если одежда персонажа должна шевелиться, то во время создания ключей анимации постарайтесь учесть и движения одежды. Но тут есть один интересный нюанс — движение одежды окажется слишком однообразным. Если походка человека должна быть постоянна, но желательно уникальной для каждого человека, то движение одежды лучше сделать разнообразным. Можно для каждого шага сделать несколько ключей анимации, в которых будут созданы разные движения одежды, но это будут уж слишком большие расходы памяти.

А если вспомнить про существование волос, которые тоже нельзя анимировать с помощью скелета, то появляется еще одна проблема, которую нужно решать. Наверно именно поэтому большинство программистов любят использовать только вершинную анимацию, чтобы не усложнять себе жизнь различными методами. Но есть еще один вариант, который тоже встречается достаточно часто — монстров можно не одевать, персонажей одеть в обтягивающую одежду, а глаза и волосы можно спрятать под шлемом или под очками с шапкой. В этом случае достаточно будет только скелета, и если не анимировать губы, то можно обойтись без вершинной анимации. Хотя если шлем хороший, то губ будет не видно и еще одна проблема с плеч долой.

Упрощать себе жизнь — хорошо, а детализация тоже неплохо. Можно пойти на компромисс — создать всех персонажей игры упрощенными (минимум вершинной анимации, а то и вовсе без нее), где все детали спрятаны и волосы, губы, глаза не видны, а также пару персонажей сделать более детализированными. Уже были подобные прецеденты и большинство компьютерных изданий и просто пользователей восхищалось детализацией, хотя она была далеко не у всех, а точнее, у единиц.

ГЛАВА 7



Программирование звука

По звуку можно писать отдельную книгу, потому что это достаточно большая тема. Мы ограничимся только основами. Нет, мы не будем говорить о том, как писать музыку и какая она должна быть. Это совершенно другая проблема, которая должна решаться профессионалами в музыке, а не программистами. Мои же познания в этой сфере ограничиваются только знаниями нот. Жанр в игре может быть любым (как электронная музыка или рок, так и живая органная музыка), главное, чтобы звуковая составляющая гармонировала с видеорядом и происходящими действиями на экране монитора.

В DirectX за работу со звуком отвечают два основных компонента: DirectMusic и DirectSound. Они состоят из множества вспомогательных интерфейсов, и в данной главе нам предстоит разобраться с основами.

Компоненты DirectMusic и DirectSound предназначены для воспроизведения звуковых данных WAV, позволяют использовать эффекты и управлять 3D-звучанием. Так в чем же разница? Первый из них предоставляет больше возможностей, например, помимо воспроизведения файлов WAV он может проигрывать MIDI, и предоставляет больше возможностей по контролю. Но, несмотря на это, DirectSound еще может применяться там, где это необходимо. К тому же, у него есть одна возможность, которой нет у DirectMusic — возможность записи звука. Да и управление 3D-звучанием в DirectMusic происходит через DirectSound, поэтому лучше знать оба этих компонента. Лучше, но не обязательно.

Итак, давайте более подробно познакомимся с возможностями этих двух компонентов и перейдем непосредственно к программированию звука.

7.1. Введение в звук

До появления DirectX для работы со звуком в ОС Windows была библиотека MMSystem (Multimedia System). Она предоставляет базовые возможности, и с

ее использованием воспроизводятся звуки в самой ОС. Проблема MMSystem заключается в том, что она должна работать со всем разнообразием звуковых карт и при этом не должна иметь прямого доступа к оборудованию. Именно поэтому в составе DirectX должно было появиться что-то более мощное.

В MSDN при рассмотрении основ графических компонентов DirectX обязательно есть ссылка на то, что там поддерживается два уровня абстракции: HAL и HEL. В случае со звуком то же самое, эти уровни существуют, давайте посмотрим зачем.

HAL (Hardware Abstraction Layer или уровень абстракции оборудования) позволяет абстрагироваться от звуковой и вне зависимости от железки использовать все "навороченные" возможности. Например, драйвер встроенной в чипсет звуковой карты может требовать одни функции для создания какого-то эффекта, а драйвер звуковой карты от Creative другие функции. Благодаря DirectSound нас эти функции не волнуют. Достаточно использовать API из DirectSound, а он уже сам заставит звук литься так, как надо.

HEL (Hardware Emulation Layer или уровень эмуляции оборудования) позволяет эмулировать возможности, которые аппаратно не поддерживаются железом. Таким образом, вне зависимости от железки, которая установлена на компьютере пользователя, программа будет работать корректно.

Эмуляция — это достаточно образная вещь. Конечно же, если звуковая карта не поддерживает стереозвучание, то его и не будет. Просто программа будет думать, что оно есть и сможет работать на два фронта (канала), хотя реально из колонок будет идти монозвук.

Но в некоторых случаях эмуляция позволяет заставить железо работать так, как оно не умеет. Например, если звуковая карта не поддерживает аппаратное MIDI, то DirectSound позволяет сделать это программно. Правда, такая эмуляция достигается достаточно большой нагрузкой на процессор. А кто говорил, что все прекрасно? Нечего использовать звук, встроенный в чипсет, простой Creative сейчас стоит не так уж и дорого, а по возможностям превосходит в несколько раз.

Компонент DirectSound позволяет воспроизводить звук с маленькой задержкой и предоставляет приложениям высокоуровневый контроль над аппаратными ресурсами. С использованием DirectSound вы можете:

- воспроизводить звуковые данные в WAV-формате;
- воспроизводить несколько звуков одновременно, что для игр является очень важной возможностью, ведь одновременно могут говорить несколько персонажей, играть музыка, плескаться вода и греметь выстрелы орудия. Мы должны иметь удобную возможность воспроизведения нескольких звуков одновременно;

- располагать звуковые данные в настраиваемом 3D-окружении;
- использовать такие эффекты, как эхо или хор. При этом вы можете динамически изменять их параметры;
- захватывать звуковые данные с входного устройства.

Тут необходимо отметить, что компонент работает в любой ОС, но при этом некоторая функциональность работает только в Windows XP.

Как мы уже знаем, DirectMusic предоставляет больше возможностей. С использованием этого компонента вы можете:

- загружать и воспроизводить звуковые данные в форматах MIDI или WAV;
- воспроизводить звук одновременно из нескольких источников;
- планировать время музыкальных событий с высокой точностью;
- программно направлять MIDI-события, такие как изменение темпа и др.;
- использовать загружаемые звуки (Downloadable Sounds, DLS);
- располагать звук в 3D-окружении;
- легко устанавливать такие эффекты, как реверберация и тональность;
- использовать более 16 MIDI-каналов. Компонент DirectMusic позволяет воспроизводить столько звуков одновременно, сколько поддерживается оборудованием;
- захватывать MIDI-данные.

Мы будем больше внимания уделять DirectMusic. Основное его преимущество для игр — возможность воспроизводить MIDI-файлы. Эти файлы занимают минимум места и позволяют воспроизводить музыку высокого качества.

Работа со звуком отличается от работы с графикой, хотя и то и другое входит в один большой пакет DirectX. Не знаю, почему разработчики плодят такое разнообразие в использовании, но надеюсь, у них были на то серьезные причины. Отличие бросается в глаза уже с первого взгляда, потому что даже экземпляры интерфейсов приходится создавать по-другому. Чаще всего для этого используется функция `CoCreateInstance`. Для работы со звуком необходимо подключить заголовочный файл dmusici.h.

Для воспроизведения музыки нам понадобятся три интерфейса: `IDirectMusicLoader8`, `IDirectMusicPerformance8` и `IDirectMusicSegment8`. Давайте рассмотрим каждый из них, и для чего они используются.

7.1.1. *IDirectMusicLoader*

Интерфейс `IDirectMusicLoader8` используется для поиска, перечисления, кэширования и загрузки объектов. Благодаря кэшированию объекты будут

загружаться не более одного раза. Это очень удобно, ведь некоторые звуки могут существовать на разных игровых уровнях, и при переходах повторная загрузка будет излишней.

Приложению достаточно иметь один экземпляр этого интерфейса для загрузки звуковых данных. Нет смысла заводить несколько экземпляров, потому что это лишние расходы памяти. К тому же, если будет несколько экземпляров, то они не смогут эффективно кэшировать данные.

В играх загрузчик используется в основном только для загрузки данных, а такие возможности, как перечисление файлов в указанной директории оказываются не востребованными.

Для создания экземпляра этого интерфейса необходимо использовать функцию CoCreateInstance. Например, экземпляр интерфейса можно создать следующим образом:

```
CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
    IID_IDirectMusicLoader8, (void**)&pLoader);
```

В качестве первого параметра (идентификатор класса CLSID) необходимо указать CLSID_DirectMusicLoader. Четвертый параметр, который должен указывать на интерфейс, используемый для коммуникации с объектом, должен быть равен IID_IDirectMusicLoader8. Последний параметр — это указатель на переменную, которая будет проинициализирована. После выполнения функции CoCreateInstance эта переменная будет указывать на созданный экземпляр.

Интерфейс DirectMusicLoader упрощает загрузку звуковых файлов с помощью следующих трех методов: SetSearchDirectory, EnumObject и ScanDirectory. Давайте посмотрим на эти методы поближе:

- SetSearchDirectory — этот метод устанавливает путь для поиска файловых объектов. Этот путь может быть указан для одного или нескольких путей;
- ScanDirectory — метод просматривает каталог на наличие файлов запрошенного класса или имеющие указанное расширение. Для каждого найденного файла вызывается метод ParseDirectory, для получения имени и идентификатора объекта. Полученная информация сохраняется во внутренней базе данных. Однажды просканировав каталог, все файлы запрошенного типа становятся доступны методу EnumObject;
- EnumObject — этот метод перечисляет все доступные объекты запрошенного типа. Тип и каталог устанавливаются с помощью метода SetSearchDirectory.

Следующие два метода повышают скорость загрузки:

- `CacheObject` — в качестве параметра метод получает объект, который необходимо кэшировать. В результате этого интерфейс гарантирует, что объект не будет загружен дважды, а значит, на загрузку не будет потрачено драгоценное время;
- `EnableCache` — с помощью этого метода можно разрешить или запретить автоматическое кэширование;
- `ClearCache` — с помощью этого метода можно очистить кэш.

7.1.2. *IDirectMusicPerformance*

Интерфейс `IDirectMusicPerformance` используется для управления проигрыванием. Если необходимо проигрывать более одного файла одновременно, то вы должны создать соответствующее количество экземпляров интерфейса `IDirectMusicPerformance`.

Для создания экземпляра интерфейса используется функция `CoCreateInstance`, например, как показано в следующей строке кода:

```
CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
    IID_IDirectMusicPerformance8, (void**)&pPerformance);
```

В качестве первого параметра необходимо указать идентификатор класса `CLSID_DirectMusicPerformance`. Четвертый параметр, который должен указывать на интерфейс, используемый для коммуникации с объектом, должен быть равен `IID_IDirectMusicPerformance8`. Последний параметр — это указатель на переменную, которая будет проинициализирована.

Среди полезных функций, которые вам могут потребоваться, можно выделить:

- `CreateAudioPath` — метод создает аудиодорожку, основываясь на указанном объекте конфигурации;
- `CreateStandardAudioPath` — создает стандартную аудиодорожку, на основе указанных параметров. Этот метод мы рассмотрим в *разд. 7.4*;
- `GetDefaultAudioPath` — метод возвращает аудиодорожку по умолчанию;
- `SetDefaultAudioPath` — метод устанавливает и делает активной дорожку по умолчанию для объекта исполнения;
- `IsPlaying` — метод определяет, началось ли воспроизведение указанного сегмента;
- `GetSegmentState` — метод определяет состояние воспроизводящегося в данный момент сегмента. Результат будет представлен в виде интерфейса `IDirectMusicSegmentState`;

- PlaySegment — метод позволяет воспроизвести сегмент;
- PlaySegmentEx — воспроизводит сегмент с указанными опциями;
- Stop — останавливает воспроизведение сегмента;
- AddPort — добавляет порт интерфейсу исполнения. Этот метод действителен только для тех интерфейсов, которые не используют аудиодорожки. В качестве параметра передается интерфейс порта IDirectMusicPort, который нужно добавить;
- RemovePort — удаляет порт из интерфейса исполнения. В качестве параметра передается интерфейс порта IDirectMusicPort, который нужно удалить;
- InitAudio — инициализирует аудио, и если необходимо, то создать аудиодорожку;
- CloseDown — прекращает работу интерфейса исполнения.

Это далеко не весь список методов, которые есть у данного интерфейса. Некоторые из них мы рассмотрим на практике в последующих разделах данной главы.

7.1.3. *IDirectMusicSegment8*

Интерфейс IdirectMusicSegment8, используется для отображения законченного сегмента звуковых данных. Этот интерфейс создается в момент загрузки данных с помощью интерфейса IDirectMusicLoader8 и его метода LoadObjectFromFile.

Основные методы этого интерфейса позволяют:

- InitPlay — инициализировать воспроизведение;
- GetAudioPathGonfig — получить объект, который отображает конфигурацию аудиодорожки, встроенной в аудиосегмент;
- Download — загрузить данные в интерфейс исполнения и загрузить аудиодорожку;
- Upload — выгрузить данные инструментов из интерфейса исполнения и выгрузить аудиодорожку;
- GetLength — получить длину сегмента;
- SetLength — установить длину сегмента в музыкальных единицах.

7.2. Воспроизведение

Давайте рассмотрим работу интерфейса IDirectMusic на простейшем примере. Это позволит нам лучше понять его работу и при дальнейшем рассмотре-

нии интерфейса отталкиваться от реального примера. Итак, для начала сделаем так, чтобы после загрузки движка в фоне начала играть музыка из MIDI-файла.

Для начала в классе движка объявляем три переменные, которые будут использоваться для хранения необходимых нам интерфейсов `IDirectMusicLoader8`, `IDirectMusicPerformance8` и `IDirectMusicSegment8`:

```
IDirectMusicLoader8* pLoader;
IDirectMusicPerformance8* pPerformance;
IDirectMusicSegment8* pSegment;
```

Теперь в конструкторе движка пишем код, показанный в листинге 7.1.

Листинг 7.1. Простой пример воспроизведения

```
// Инициализация IDirectMusicLoader и IDirectMusicPerformance
CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
    IID_IDirectMusicLoader8, (void**)&pLoader);
CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
    IID_IDirectMusicPerformance8, (void**)&pPerformance);

// Инициализация аудио
pPerformance->InitAudio(NULL, NULL, NULL,
    DMUS_APATH_SHARED_STEREOPLUSREVERB, 64, DMUS_AUDIOF_ALL, NULL);

// Загрузка звуковых данных
if (FAILED(pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
    IID_IDirectMusicSegment8, L"Media\\game.mid",
    (LPVOID*) &pSegment)))
 pSegment = NULL;

// Загрузить инструменты
pSegment->Download(pPerformance);

// Проиграть сегмент
pPerformance->PlaySegmentEx(pSegment, NULL, NULL, 0, 0,
    NULL, NULL, NULL);
```

Это минимально необходимый и достаточный для воспроизведения музыкального файла код. В самом начале инициализируются интерфейсы `IDirectMusicLoader8` и `IDirectMusicPerformance8`.

После вызываем метод `InitAudio` интерфейса `IDirectMusicPerformance8`. Этот метод используется для инициализации интерфейса и должен вызываться до того, как вы начали воспроизведение. Раньше для инициализации использо-

вался метод Init, а теперь необходимо вызывать именно InitAudio, который в общем виде выглядит следующим образом:

```
HRESULT InitAudio(
  IDirectMusic** ppDirectMusic,
  IDirectSound** ppDirectSound,
  HWND hWnd,
  DWORD dwDefaultPathType,
  DWORD dwPChannelCount,
  DWORD dwFlags,
  DMUS_AUDIOPARAMS *pParams
);
```

Здесь имеется аж семь параметров и не помешало бы узнать, для чего они нужны:

- ppDirectMusic — указатель на объект IDirectMusic или IDirectMusic8. Если через этот параметр передать действительный указатель, то существующий объект будет назначен нашему объекту IDirectMusicPerformance8. Если передать здесь нулевое значение, то будет создан новый экземпляр интерфейса IDirectMusic. Если вам понадобится этот указатель, то его можно будет потом получить с помощью метода QueryInterface;
- ppDirectSound — указатель на интерфейс IDirectSound, который необходимо использовать. Если указатель нулевой, то будет создан новый экземпляр;
- hWnd — окно, которое должно использоваться для создания DirectSound. Если этот параметр нулевой, то будет использоваться активное в данный момент окно;
- dwDefaultPathType — параметр определяет формат аудиодорожки по умолчанию. В этом параметре можно указать одно из следующих значений:
  - DMUS_APATH_DYNAMIC_3D — одна шина к 3D-буферу;
  - DMUS_APATH_DYNAMIC_MONO — одна шина к монобуферу;
  - DMUS_APATH_DYNAMIC_STEREO — две шины к стереобуферу;
  - DMUS_APATH_SHARED_STEREOPLUSREVERB — обычная звуковая настройка с использованием стереовыхода и реверберации.
- dwPChannelCount — количество каналов воспроизведения;
- dwFlags — флаги, через которые можно задать требуемые возможности. Здесь можно указать одно из следующих значений:
  - DMUS_AUDIOF_3D — 3D-буфер. Этот параметр не имеет особого значения, потому что буферы в 3D-аудиоканалах всегда используют эту поддержку;

- DMUS_AUDIOF_ALL — параметр использует все возможности;
- DMUS_AUDIOF_BUFFERS — применяются составные буферы;
- DMUS_AUDIOF_EAX — обеспечивается поддержка Environmental Audio Extensions (EAX, аудиорасширения окружения);
- DMUS_AUDIOF_STREAMING — поддерживается в поточном звучании.

- pParams — через этот параметр задаются параметры синтезатора и определяются установленные параметры. Можно указать нулевое значение, чтобы использовать параметры по умолчанию.

Как видите, в большинстве параметров можно указывать нулевые значения. В нашем примере, я думаю, как и в большинстве ваших, будут тоже значения по умолчанию:

```
pPerformance->InitAudio(NULL, NULL, NULL,
   DMUS_APATH_SHARED_STEREOPLUSREVERB, 64, DMUS_AUDIOF_ALL, NULL);
```

Теперь мы готовы загружать аудиоданные. Для этого можно использовать метод LoadObjectFromFile интерфейса IDirectMusicLoader8. Вы должны использовать этот метод вместо GetObject, если данные находятся в файле. В общем виде загрузка выглядит следующим образом:

```
HRESULT LoadObjectFromFile(
  REFGUID rguidClassID,
  REFIID iidInterfaceID,
  WCHAR *pwzFilePath,
  void ** ppObject
);
```

Метод загружает звуковые данные, создает экземпляр класса IDirectMusicSegment8 и помещает загруженные данные в него.

Тут имеется всего четыре параметра:

- rguidClassID — уникальный идентификатор класса IDirectMusicSegment8, здесь необходимо указывать константу CLSID_DirectMusicSegment;
- iidInterfaceID — идентификатор интерфейса. Необходимо использовать IID_IDirectMusicSegment8;
- pwzFilePath — путь к файлу, который необходимо загрузить;
- ppObject — указатель на переменную, в которой мы получим результат, т. е. экземпляр IDirectMusicSegment8 с загруженными звуковыми данными.

Теперь необходимо выполнить еще одно подготовительное действие — загрузить звуковые данные в интерфейс IDirectMusicPerformance8:

```
pSegment->Download(pPerformance);
```

Загрузка происходит с помощью метода Download, которому необходимо передать указатель на IDirectMusicPerformance8, куда должны загружаться данные.

Все, подготовка закончена и можно переходить непосредственно к воспроизведению. Для этого используется метод PlaySegment или PlaySegmentEx. Второй предоставляет больше параметров, поэтому рассмотрим его:

```
HRESULT PlaySegmentEx(
 IUnknown* pSource,
 WCHAR *pwzSegmentName,
 IUnknown* pTransition,
 DWORD dwFlags,
 __int64 i64StartTime,
 IDirectMusicSegmentState** ppSegmentState,
 IUnknown* pFrom,
 IUnknown* pAudioPath
);
```

Не могу вас оставить один на один с таким большим количеством параметров, поэтому давайте рассмотрим их:

- pSource — источник данных, из которого нужно воспроизвести данные;
- pwzSegmentName — по идее, здесь должно было быть имя сегмента, но пока этот параметр зарезервирован и должен быть равен нулю;
- pTransition — интерфейс, который используется для модуляции;
- dwFlags — флаги, определяющие поведение;
- i64StartTime — позиция, начиная с которой нужно воспроизводить данные из сегмента;
- ppSegmentState — указатель на интерфейс IDirectMusicSegmentState, который отображает состояние проигрываемого сегмента. Этот параметр может быть равен нулю;
- pFrom — здесь можно указать сегмент, который должен быть проигран перед завершением воспроизведения данного сегмента;
- pAudioPath — интерфейс, который определяет аудиодорожку, через которую нужно воспроизводить. Параметр может быть равен нулю, чтобы использовать дорожку по умолчанию.

У метода PlaySegmentEx только первый параметр является обязательным, где вы указываете сегмент для воспроизведения. Все остальные параметры могут быть нулевыми. В нашем примере воспроизведение запускается следующим образом:

```
pPerformance->PlaySegmentEx(pSegment, NULL, NULL, 0, 0,
   NULL, NULL, NULL);
```

7.3. Завершение воспроизведения

После завершения работы программы можно все бросить и ничего не делать, но лучше использовать корректное завершение. Что это значит? Конечно же, необходимо освободить всю выделенную память. Но если в данный момент воспроизводится сегмент звуковых данных, то его нужно остановить и закрыть интерфейс воспроизведения. В деструкторе добавляем следующий код:

```
pPerformance->StopEx(NULL, NULL, 0);
pPerformance->CloseDown();

if (pSegment){pSegment->Release();pSegment=NULL;}
if (pLoader){pLoader->Release();pLoader =NULL;}
if (pPerformance){pPerformance->Release();pPerformance=NULL;}
```

Для того чтобы остановить воспроизведение, необходимо использовать метод `Stop` или `StopEx`. Рассмотрим второй из них:

```
HRESULT StopEx(
  IUnknown *pObjectToStop,
  __int64 i64StopTime,
  DWORD dwFlags
);
```

Здесь имеется три параметра:

- `pObjectToStop` — интерфейс сегмента, воспроизведение которого нужно остановить;
- `i64StopTime` — время, в которое нужно остановить воспроизведение;
- `i64StopTime` — флаги, которые определяют время остановки.

Чтобы остановить воспроизведение текущего сегмента немедленно, достаточно установить все параметры нулевыми.

После остановки воспроизведения закрываем интерфейс с помощью метода `CloseDown`. Этому методу никакие параметры не нужны. Вот теперь освобождаем память, уничтожая все созданные экземпляры интерфейсов.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter7\Sound.

7.4. Погружение в 3D

Я не такой уж и старый, но прекрасно помню те времена, когда мы радовались даже монозвучанию магнитофонов "Весна". Потом появились более со-

временные варианты той же "Весны", но с двумя колонками и настоящим стерео. Но самый большой шок я испытал, когда мой брат привез из ФРГ (для тех, кто не знаком с такой страной, раньше Германия делилась на ФРГ и ГДР) плеер Sony Walkman. Его звучание было божественным. Я даже не думал, что такое можно записать на магнитную ленту, а потом воспроизвести с таким великолепным качеством и с помощью такой маленькой коробочки.

С тех пор прошло более 15 лет и в звучании произошло не так уж и много изменений, но они значительны. Например, появились технологии, позволяющие воспроизводить уже не 2D-звук, а самый настоящий 3D, правда здесь уже двумя колонками не отделаешься.

В играх погружение в 3D дает нам множество преимуществ и приятных удобств. Не помню уже игру, в которой я впервые ощутил преимущества от использования стереозвука (кажется Heretic), но именно тогда я понял, что звук не только улучшает ощущения от игры, но и упрощает прохождение. Допустим, что вы идете по коридорам 3D виртуального мира и слышите шаги справа от себя. Впереди большая комната и уже не сложно догадаться, что в этой комнате за правым углом находится противник. Готовим боеголовку помощнее и, выскакивая из-за угла, разносим все, что только попадается под руку.

Давайте на примере рассмотрим использования 3D-звука. Для того чтобы можно было управлять источником звука и положением слушателя, необходимо для начала изменить инициализацию. Новый код инициализации звука, который необходимо написать в конструкторе движка игры, показан в листинге 7.2.

Листинг 7.2. Инициализация 3D-звука

```
// Инициализация IDirectMusicLoader и IDirectMusicPerformance
CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
     IID_IDirectMusicLoader8, (void**)&pLoader);
CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
     IID_IDirectMusicPerformance8, (void**)&pPerformance);

// Инициализация аудио
pPerformance->InitAudio(NULL, NULL, hWnd, DMUS_APATH_DYNAMIC_STEREO,
     128, DMUS_AUDIOF_ALL, NULL);

// Создание стандартной аудиодорожки
if (FAILED(pPerformance->CreateStandardAudioPath(
    DMUS_APATH_DYNAMIC_3D, 64, TRUE, &pAudioPath)))
 pAudioPath = NULL;
```

```
// Получение интерфейса звукового буфера
if (FAILED(pAudioPath->GetObjectInPath(0, DMUS_PATH_BUFFER, 0,
      GUID_NULL, 0, IID_IDirectSound3DBuffer, (LPVOID*) &p3DBuffer)))
 p3DBuffer = NULL;

// Получение текущих параметров буфера
DS3DBUFFER dsBufferParams;
dsBufferParams.dwSize = sizeof(DS3DBUFFER);
p3DBuffer->GetAllParameters(&dsBufferParams);

// Установка новых параметров буфера
dsBufferParams.dwMode = DS3DMODE_HEADRELATIVE;
dsBufferParams.flMinDistance = 5;
dsBufferParams.flMaxDistance = 500;
p3DBuffer->SetAllParameters(&dsBufferParams, DS3D_IMMEDIATE);

// Получение интерфейса слушателя
DS3DLISTENER dsListenerParams;
if (FAILED(pAudioPath->GetObjectInPath( 0, DMUS_PATH_PRIMARY_BUFFER, 0,
     GUID_NULL, 0, IID_IDirectSound3DListener, (LPVOID*) &pListener)))
  pListener=NULL;

// Получение параметров слушателя
dsListenerParams.dwSize = sizeof(DS3DLISTENER);
pListener->GetAllParameters(&dsListenerParams);

// Установка параметров
dsListenerParams.flDopplerFactor = 0;
dsListenerParams.flRolloffFactor = 0;
pListener->SetAllParameters(&dsListenerParams, DS3D_IMMEDIATE);

// Загрузка данных
if (FAILED(pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
   IID_IDirectMusicSegment8, L"Media\\game.mid", (LPVOID*) &pSegment)))
 pSegment = NULL;

// Воспроизвести
pSegment->Download(pPerformance);
pPerformance->PlaySegmentEx(pSegment, NULL, NULL, DMUS_SEGF_BEAT,
     0, NULL, NULL, pAudioPath);
```

Этот код можно было бы сократить, но я решил показать вам некоторые необязательные, но полезные настройки.

В самом начале мы инициализируем интерфейсы IDirectMusicLoader и IDirectMusicPerformance. Ничего нового тут нет. После этого инициализируем аудио с помощью метода InitAudio. Тут также ничего нового. Новое начинается со следующей строки, где вызывается метод CreateStandardAudioPath для создания стандартной аудиодорожки. В общем виде этот метод выглядит следующим образом:

```
HRESULT  CreateStandardAudioPath(
  DWORD dwType,
  DWORD dwPChannelCount,
  BOOL  fActivate,
  IDirectMusicAudioPath **ppNewPath
);
```

Здесь имеется всего четыре параметра:

- dwType — тип аудиодорожки. В этом параметре можно указать одно из четырех значений:
  - DMUS_APATH_DYNAMIC_3D — одна шина к 3D-буферу;
  - DMUS_APATH_DYNAMIC_MONO — одна шина к монобуферу;
  - DMUS_APATH_DYNAMIC_STEREO — две шины к стереобуферу;
  - DMUS_APATH_SHARED_STEREOPLUSREVERB — обычная звуковая настройка с использованием стереовыхода и реверберации;
- dwPChannelCount — параметр определяет количество каналов;
- fActivate — если этот параметр равен true, то буфер будет активный после создания;
- ppNewPath — переменная типа IDirectMusicAudioPath, которая после выполнения метода будет содержать созданную аудиодорожку.

Параметр dwType принимает те же значения, что и четвертый параметр у метода InitAudio, который мы рассмотрели в *разд. 7.2*. В данном случае, мы рассматриваем 3D-звук, поэтому в этом параметре указали значение DMUS_APATH_DYNAMIC_3D.

Вот тут необходимо заметить, почему в описании типа DMUS_APATH_DYNAMIC_3D, приведенного ранее, написано "одна шина"? Дело в том, что 3D-дорожка требует, чтобы звук был моно. Даже если аудиоданные будут стерео, они автоматически будут приведены в вид моно. Почему? Давайте рассмотрим на примере — допустим, что источник звука находится справа от нас. Человеческое ухо не сможет выделить стереозвучание, потому что в правое ухо звук будет попадать раньше, а в левое позже и в основном отраженный сигнал.

Стереозвучание будет ощущаться только в том случае, если вы находитесь между двумя источниками звука. Поэтому для воспроизведения фоновой музыки, которая создает атмосферу игры, используется стереодорожка, а для воспроизведения эффектов, звуков шагов и стрельбы должен использоваться 3D-буфер.

Новая дорожка создана, теперь необходимо получить ее буфер. Для этого вызываем метод GetObjectInPath. В общем виде он выглядит следующим образом:

```
RESULT GetObjectInPath(
  DWORD dwPChannel,
  DWORD dwStage,
  DWORD dwBuffer,
  REFGUID guidObject,
  DWORD dwIndex,
  REFGUID iidInterface,
  void ** ppObject
);
```

Здесь имеется всего семь параметров:

- dwPChannel — канал воспроизведения;
- dwStage — фаза аудиодорожки. Здесь мы указываем параметр DMUS_PATH_BUFFER, чтобы получить буфер;
- dwBuffer — индекс буфера;
- guidObject — GUID-идентификатор класса. Параметр может игнорироваться, и мы можем указать GUID_NULL;
- dwIndex — индекс искомого объекта. Если параметр равен нулю, то будет возвращен первый найденный;
- iidInterface — искомый интерфейс;
- iidInterface — переменная для сохранения результата, т. е. запрошенного интерфейса.

Теперь у нас есть аудиобуфер и необходимо выполнить пару настроек над буфером. Но чтобы произвести эти настройки, необходимо получить текущие параметры в виде структуры DS3DBUFFER, после чего изменяем нужный параметр и устанавливаем их.

Итак, для управления параметрами буфера необходима структура DS3DBUFFER, которая выглядит следующим образом:

```
typedef struct {
  DWORD      dwSize;
  D3DVECTOR  vPosition;
```

```
  D3DVECTOR  vVelocity;
  DWORD      dwInsideConeAngle;
  DWORD      dwOutsideConeAngle;
  D3DVECTOR  vConeOrientation;
  LONG       lConeOutsideVolume;
  D3DVALUE   flMinDistance;
  D3DVALUE   flMaxDistance;
  DWORD      dwMode;
} DS3DBUFFER, *LPDS3DBUFFER;
```

Давайте посмотрим параметры этой структуры:

- dwSize — корректный размер структуры;
- vPosition — вектор позиции источника звука;
- vVelocity — вектор скорости источника звука, на основе которого рассчитывается 3D-звук;
- dwInsideConeAngle — внутренний угол конуса, определяющего распределение излучения звука;
- dwOutsideConeAngle — внешний угол конуса, определяющего распределение излучения звука;
- vConeOrientation — направление ориентации конуса;
- lConeOutsideVolume — громкость звука за пределами конуса;
- flMinDistance — минимальная дистанция, в пределах которой звук неизменен;
- flMaxDistance — максимальная дистанция распределения звука;
- dwMode — здесь может быть одно из трех значений:
  - DS3DMODE_DISABLE — обработка 3D-звука отключена;
  - DS3DMODE_HEADRELATIVE — в таком режиме изменение положение буфера влияет на положение слушателя;
  - DS3DMODE_NORMAL — нормальный режим.

Изменение некоторых параметров этой структуры возможно и с помощью специализированных методов. Некоторые из них мы увидим чуть позже.

Создание 3D-звука в DirectSound основано на эффекте Доплера (Doppler), который рассчитывает смещение каждого буфера или слушателя, которые имеют скорость. Чтобы звук был более реалистичным, вы должны рассчитать скорость движения слушателя или источника.

После настройки параметров буфера (источника звука) мы настраиваем слушателя. В данном примере слушатель не используется, но для иллюстрации возможностей код настройки есть.

За слушателя отвечает интерфейс `IDirectSound3DListener`, который мы можем получить с помощью метода `GetObjectInPath`:

```
if (FAILED(pAudioPath->GetObjectInPath(0, DMUS_PATH_PRIMARY_BUFFER,
   0, GUID_NULL, 0, IID_IDirectSound3DListener,
   (LPVOID*) &pListener)))
 pListener=NULL;
```

За настройки слушателя отвечает структура `DS3DLISTENER`, которая выглядит следующим образом:

```
typedef struct {
  DWORD       dwSize;
  D3DVECTOR   vPosition;
  D3DVECTOR   vVelocity;
  D3DVECTOR   vOrientFront;
  D3DVECTOR   vOrientTop;
  D3DVALUE    flDistanceFactor;
  D3DVALUE    flRolloffFactor;
  D3DVALUE    flDopplerFactor;
} DS3DLISTENER, *LPDS3DLISTENER;
```

Хотя свойства слушателя можно изменять с помощью специализированных методов, давайте посмотрим параметры этой структуры, они вам пригодятся:

- `dwSize` — корректный размер структуры;
- `vPosition` — вектор позиции слушателя;
- `vVelocity` — вектор скорости слушателя;
- `vOrientFront` — вектор, определяющий направление взгляда, т. е. куда смотрит лицо слушателя;
- `vOrientTop` — вектор, определяющий направление вверх, т. е. куда смотрит макушка слушателя;
- `flDistanceFactor` — количество метров в одной единице вектора;
- `flRolloffFactor` — соотношение ослабления звука от расстояния к реальному в нашем мире;
- `flDopplerFactor` — соотношение эффекта Доплера к реальному в нашем мире. Значение этого параметра может быть от 0.0 до 10.0.

После выполнения всех настроек загружаются звуковые данные и начинается воспроизведение. Обратите внимание, как именно начинается воспроизведение:

```
pPerformance->PlaySegmentEx(pSegment, NULL, NULL, DMUS_SEGF_BEAT,
   0, NULL, NULL, pAudioPath);
```

Самое интересное — это последний параметр. Чтобы использовать настроенную нами ранее звуковую дорожку, мы указываем ее в качестве последнего параметра.

7.5. Контроль над 3D

У нас создан буфер и запущена музыка. Давайте теперь посмотрим, как можно изменять параметры звучания, а именно положение источника звука и слушателя. В нашем примере будет изменяться только источник звука. Давайте сделаем эффект, как будто звук исходит из сетки Mesh, загруженной в сцену. Таким образом, если вы будете бегать по миру вокруг фигуры, звук будет бегать вокруг вас.

Для реализации данного примера вполне логично поместить изменение положения источника звука в функцию KeyControl, где мы изменяем положение камеры, в зависимости от выполняемых пользователем действий. Итак, в конце функции пишем следующий код:

```
p3DBuffer->SetPosition(pos.x, pos.y, pos.z, DS3D_IMMEDIATE);
p3DBuffer->SetVelocity(pos.x, pos.y, pos.z, DS3D_IMMEDIATE);
```

Всего две строки решают проблему. С помощью метода SetPosition устанавливаем позицию источника звука, а с помощью SetVelocity устанавливаем скорость. В качестве первых трех параметрах этих методов передаются координаты. Последний параметр — это флаг, который определяет, когда должны вступить изменения в силу. Чтобы это произошло немедленно, укажите флаг DS3D_IMMEDIATE.

Изменение параметров слушателя происходит примерно тем же способом. У интерфейса IDirectSound3DListener тоже есть методы SetPosition и SetVelocity:

```
pListener->SetPosition(pos.x, pos.y, pos.z, DS3D_IMMEDIATE);
pListener->SetVelocity(pos.x, pos.y, pos.z, DS3D_IMMEDIATE);
pListener->SetOrientation(vecFB.x, vecFB.y, vecFB.z,
          up.x, up.y, up.z, DS3D_IMMEDIATE);
```

В третьей строке устанавливаем ориентацию головы. Для этого используем метод SetOrientation. У метода шесть параметров — первые три определяют координаты вектора направления вперед, а последние три параметра определяют направление макушки.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter7\3DSound.

7.6. Мультизвук

В игре обязательно должна воспроизводиться музыка, причем не только одна мелодия или единственный звуковой файл. Звуковое сопровождение в игре можно разделить на две части — фон и передний план. В фоне всегда должна играть музыка, которая создает настроение, в соответствии с текущей сценой, а на переднем плане воспроизводятся эффекты, такие как звуки от шагов противника или выстрелов. Причем фоновая музыка, шаги и множество выстрелов должны звучать одновременно.

Для того чтобы два звуковых потока воспроизводились одновременно, нет необходимости делать что-то сверхъестественное. Достаточно только установить один флаг, и все готово.

Итак, в заголовочном файле GraphEngine.h добавляем переменную для хранения второго сегмента:

```
IDirectMusicSegment8* pSegment;
IDirectMusicSegment8* pSegment2;
```

Теперь посмотрим, как будет происходить загрузка и воспроизведение данных:

```
// Загрузка файла 1
if (FAILED(pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
     IID_IDirectMusicSegment8, L"Media\\game.mid",
     (LPVOID*) &pSegment)))
  pSegment = NULL;

// Загрузка файла 2
if (FAILED(pLoader->LoadObjectFromFile(CLSID_DirectMusicSegment,
     IID_IDirectMusicSegment8, L"Media\\drumpad-voc_female_ec.wav",
     (LPVOID*) &pSegment2)))
 pSegment2 = NULL;

// Загрузка сегментов данных
pSegment->Download(pPerformance);
pSegment2->Download(pPerformance);

// Воспроизведение файла 1
pPerformance->PlaySegmentEx(pSegment, NULL, NULL,
     DMUS_SEGF_BEAT, 0, NULL, NULL, pAudioPath);

// Воспроизведение файла 2
pPerformance->PlaySegmentEx(pSegment2, NULL, NULL,
     DMUS_SEGF_SECONDARY | DMUS_SEGF_DEFAULT, 0,
     NULL, NULL, NULL);
```

Загрузка файлов происходит одинаково. Разница только в воспроизведении, а если быть более точным, то в указываемых флагах. Для музыкальных данных, которые будут воспроизводиться в фоне, в качестве флага указываем DMUS_SEGF_BEAT. Здесь может быть и другой флаг, а может быть и ноль. По умолчанию такой сегмент будет восприниматься как первичный и будет заглушать воспроизводимые данные.

Для эффектов, чтобы они играли одновременно с музыкой, в качестве флага задаем DMUS_SEGF_SECONDARY. Именно этот флаг указывает на то, что поток является вторичным.

Тут необходимо заметить, что старые сегменты могут сохраняться и замусоривать память. Чтобы очистить старые данные, необходимо вызвать метод CollectGarbage интерфейса IDirectMusicLoader8:

```
pLoader->CollectGarbage();
```

Как видите, воспроизвести два файла одновременно не так уж и сложно. В случае с использованием MMSystem, при решении такой же задачи проблемы есть, причем очень серьезные.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter7\Multi.

7.7. Управление параметрами

Теперь давайте поговорим о том, как можно управлять параметрами звука. Что под этим понимается? Иногда может потребоваться изменять громкость звука, а лучше изначально заложить в игре возможность изменять громкость звука.

Громкость — это параметр, который можно изменить с помощью функции SetGlobalParam. В общем виде эта функция выглядит следующим образом:

```
HRESULT SetGlobalParam(
  REFGUID rguidType,
  void* pParam,
  DWORD dwSize
);
```

Здесь имеется всего три параметра:

- rguidType — идентификатор параметра, который вы хотите изменить. Нет, помнить все GUID нет необходимости, потому что для всех есть константы с понятными именами, например, для звука эта константа равна GUID_PerfMasterVolume;

- `pParam` — новое значение параметра, которое необходимо установить;
- `dwSize` — размер значения, указанного во втором параметре.

Тут же необходимо рассказать о двух интересных константах:

- `DMUS_VOLUME_MAX` — максимально возможное значение громкости звука;
- `DMUS_VOLUME_MIN` — минимально возможное значение громкости звука.

В играх желательно делать возможность управления звуком, потому что не всегда есть возможность управлять ими с колонок (пассивное оборудование или наушники без регулятора громкости), а люди, играющие на рабочем месте, должны иметь возможность быстро убрать громкость. Не забывайте про таких людей, особенно про секретарш, которые просиживают за линиями по 7 часов рабочего дня. Если ваша игра будет легко прятаться и будет иметь возможность регулирования звука, чтобы не засветиться перед начальством, то она сможет стать бестселлером в кругах "работяг", которых в России (да и не только в нашей стране) очень много.

Примечание

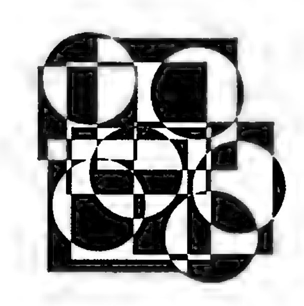
Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter7\Volume.

7.8. Резюме

При разработке игр вы не обязаны использовать DirectSound или DirectMusic. Некоторые разработчики предпочитают использовать независимую библиотеку Bbass, о которой вы можете почитать в дополнительной документации на компакт-диске из каталога Doc\Sound.

Если вы действительно создаете игру, то постарайтесь уделить звуковому сопровождению максимально возможное внимание. Если игрок входит в комнату, и музыкальное сопровождение сменяется на тревожное, потому что за углом прячется противник, то это громадный плюс вашей игре. Ну а если во время кровавой битвы заиграет музыка а-ля Цирк, то это уже сложно назвать звуковым сопровождением. Слова, которые описывают такую ситуацию, просто нельзя употреблять в книгах, их вырежет редактор ☺.

ГЛАВА 8



Игровые эффекты

В этой главе нам предстоит поговорить об игровых эффектах и некоторых трюках, которые вы можете использовать в своих будущих проектах для украшения игрового мира. Да, мир нужно украшать, чтобы он не казался скучным и однообразным. Я уверен, что вам уже через полчаса надоест бегать по серым коридорам серого мира. Украшать игру просто необходимо, и в этой главе нам предстоит узнать некоторые возможные способы.

С другой стороны, никакие эффекты не смогут сделать скучную игру бестселлером. Когда я первый раз смотрел фильм "Властелин колец", то я поражался компьютерным эффектам и созданным красотам, ландшафтам. Фильм смотрелся на одном дыхании. Вторая часть смотрелась еще лучше. Но перед просмотром третьей части я решил просмотреть все три фильма подряд и во время почти 9-ти часов просмотра телевизора я засыпал. Слишком скучно все оказалось, и даже компьютерные эффекты не смогли сгладить скуку. Мораль сей басни такова — если игра скучная, то никакие красоты не спасут, но если игра имеет хороший сюжет, то красоты смогут сделать игру еще лучше и выделить ее на фоне всех остальных.

В книгах [4] и [6] вы найдете основы создания эффектов, а в данной главе мы углубимся в эту тему и рассмотрим более сложные алгоритмы и решения. Эффекты — это вам не шубу летом носить, а творческий процесс и тут необходимо искать собственные решения, и я надеюсь, что мои решения помогут вам.

8.1. Обман зрения

С целью оптимизации графики некоторые артефакты можно и даже нужно делать двумерными. Да, мы обманем тем самым игрока, но если сделать обман аккуратным, то никто и ничего не заметит, потому что не будет обращать внимания во время динамичных баталий.

Как же происходит обман зрения в 3D-мире через 2D-объекты? Смысл в том, что вы как бы помещаете в виртуальный мир спрайты. Это можно сделать достаточно просто — создаем из четырех вершин прямоугольник и натягиваем на него спрайт в виде текстуры. Теперь достаточно только показать прямоугольник видимой стороной, где текстура отображается правильно, ведь на обратной стороне она выглядит зеркально.

Чтобы убедиться в эффективности двумерных объектов при создании 3D-сцен, я люблю приводить пример Billboard, который можно найти в каталоге DXSDK9\Samples\C++\Direct3D\Billboard (я его подробно рассматривал в книгах [4] и [6]). Запустите сейчас эту программу, и вы полетите над землей, засаженной деревьями (рис. 8.1). Красиво летим!!! А вы заметили, что деревья на самом деле двумерные и их всего-то три вида, просто рисуются они разного размера с разным оттенком цвета? Если заметили, то только потому, что в данной сцене не было врага, с которым нужно было бороться. Если бы на вас летели вражеские самолеты, то с вероятностью 90% вы не стали бы рассматривать достопримечательности на земле, когда нужно убивать противника.



Рис. 8.1. Пример программы Billboard

В данном примере использование спрайтов в качестве деревьев оправдано. Каждое дерево состоит из множества веток и тысяч листьев. Чтобы создать

все это богатство в 3D-редакторе потребуется слишком много вершин. Сетка получится слишком сложной, и достаточно накладной для воспроизведения даже с помощью самой мощной видеокарты. А теперь представим, что таких сеток/деревьев будет 1000!!! Ужас!!! В программе Billboard для ускорения перебора деревьев используется сортировка. Если убрать ее, то производительность снизится в три-четыре раза. Я думаю, если деревья рисовать в виде точек и полигонов, то производительность работы примера упадет в сотни раз.

Как я уже сказал, подобный пример я уже подробно рассмотрел в книгах [4] и [6] и писать еще один раз не имеет смысла. Я только хотел сделать акцент на использование подобной технологии отображения с точки зрения игр.

Обман зрения желательно использовать со сферическими или цилиндрическими поверхностями, где с какой бы стороны вы не посмотрели на объект, а он выглядит примерно одинаково. Раньше в играх в виде спрайтов делали такие предметы, как жизнь и орудие, но вот автомат трудно себе представить сферическим, поэтому слишком заметно было обман, когда мы бежим вокруг него, а он все время повернут к игроку боком.

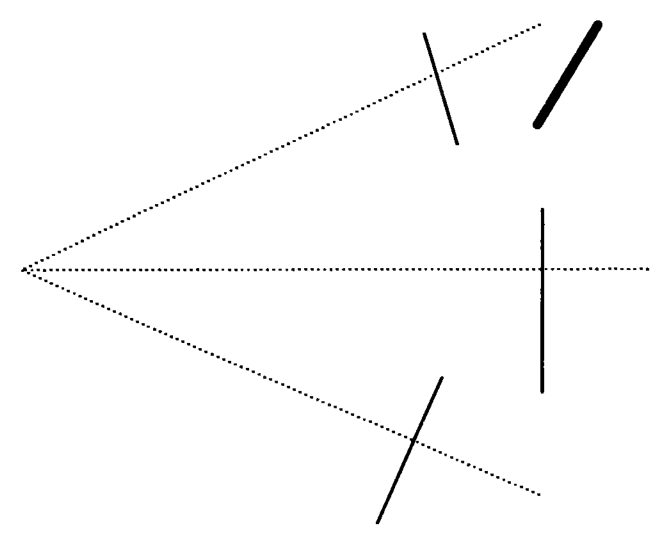


Рис. 8.2. Направление взгляда (пунктирные линии) и положение прямоугольников с текстурой (сплошные линии)

Прямоугольник с текстурой должен отображаться четко перпендикулярно по направлению взгляда, иначе текстура будет выглядеть искаженной, и игрок сразу же увидит обман. Посмотрите на рис. 8.2, где графически показано направление взгляда и отображение прямоугольников. Пунктирные линии символизируют направление взгляда, а сплошные линии — это положение пря-

моугольников. Жирная линия показывает неверное положение. Линия не перпендикулярна и картинка будет искаженной, а значит, обман быстро раскроется, а это будет неприятно.

Чтобы четко позиционировать прямоугольники, необходимо всего лишь получить инвертированную матрицу к текущей матрице просмотра. Инвертирование можно легко произвести с помощью функции `D3DXMatrixInverse`, которая в общем виде выглядит следующим образом:

```
D3DXMATRIX *D3DXMatrixInverse(
  D3DXMATRIX *pOut,
  FLOAT *pDeterminant,
  CONST D3DXMATRIX *pM
);
```

Здесь имеются всего три параметра:

- `pOut` — выходная структура, куда будет записан результат;
- `pDeterminant` — детерминанта, которую для данной операции использовать не нужно и параметр должен быть нулевым;
- `pM` — указатель на матрицу, которую необходимо инвертировать.

После инвертирования мы получаем направление, в котором должен быть установлен прямоугольник с текстурой. Останется только выставить положение, которое на данный момент идентично положению глаза (ведь мы используем для инвертирования матрицу VIEW), т. е. необходимо заполнить параметры `_41`, `_42`, `_43` матрицы, что соответствует координатам X, Y и Z.

8.2. Видео

С тех пор, как компьютер набрался мощности и смог воспроизводить видео в реальном времени, видео начали использовать и в играх. Некоторые игры были даже полностью построены на видеороликах, которые проигрывались в зависимости от действий игрока. Но в большинстве случаев разработчики ограничивались только видеороликами между уровнями.

В настоящее время видеоролики между уровнями уже используются реже, потому что разработчики стараются делать ролики на движке игры. Это позволяет сделать видео более качественным и при этом более экономным к дисковому пространству.

Для работы с видео в составе DirectX есть DirectShow. Это интерфейс программирования приложений потоковых медиаданных для платформы Microsoft. Да, DirectShow разрабатывался именно для платформы MS, а не для Linux или Apple. Что такое потоковые данные? Это не только видео, но и

аудиоданные. С помощью DirectShow ваши приложения могут отображать и захватывать видео- и аудиоданные с высоким качеством.

Среди поддерживаемых DirectShow форматов можно выделить Advanced Systems Format (ASF), Motion Picture Experts Group (MPEG), Audio-Video Interleaved (AVI), MPEG Audio Layer-3 (MP3) и др. Этот список может легко расширяться. DirectShow использует систему фильтров различного уровня, с помощью которых происходит обработка и отображение данных. Создав фильтр для обработки определенного формата и зарегистрировав его в системе, вы сможете отображать и сохранять потоковые данные в собственном формате.

Для всех ныне существующих форматов уже имеются фильтры, достаточно только установить их. Да, не все они бесплатные — есть и платные, такие как DVD, который по умолчанию не поддерживается системой, поэтому перед установкой программы на компьютер клиента убедитесь, что у него есть необходимый фильтр, или используйте только то, что устанавливается в Windows по умолчанию.

Самый простой способ узнать, сможет ли DirectShow отобразить нужный файл — это попробовать воспроизвести его с помощью Microsoft Media Player, который в своей работе также использует DirectShow и его фильтры. Если файл воспроизведен корректно, то ваша программа тоже сможет это сделать. Проблема усложняется тем, что в разных версиях DirectX и Media Player имеется разный набор фильтров.

Компонент DirectShow — это набор интерфейсов, с некоторыми из которых мы познакомимся в этом разделе. Этот компонент автоматически определяет наличие в системе ускорителей и использует их на полную мощность. Если быть точнее, то мощность использования зависит от драйвера, потому что именно через него происходит общение между ускорителем и DirectShow.

В составе DirectX SDK можно найти множество полезных примеров, с помощью которых можно быстро разобраться с технологией работы. Эти примеры расположены в каталоге DXSDK9\Samples\C++\DirectShow. Примеры просты, но понять их достаточно сложно не посвященному человеку, потому что сама технология довольно сложная и требует понимания основ работы DirectShow и фильтров. Давайте напишем небольшой пример, который будет проще предоставляемого в составе SDK и благодаря которому мы на практике познакомимся с принципами работы необходимых интерфейсов.

8.2.1. База для разработки фильтра

Разработать фильтр с нуля не такая уж и простая задача. Но всеми не любимая и повсеместно не уважаемая MS позаботилась о нас. Чтобы не говорили, но корпорация делает достаточно и для пользователей, и для разработчиков,

чтобы заслужить уважения, за что им от меня огромный respect и "uvajuha" (уважение) ☺! Но это так, небольшая пауза на вынесение заслуженной благодарности.

А теперь давайте перейдем непосредственно к примеру, который будет отображать информацию на текстуре. Пример основан на одном из примеров из состава SDK, который вы можете найти в каталоге DXSDK9\Samples \C++\DirectShow\Players\Texture3D9. Мы будем использовать этот код для изучения и для того, чтобы разобраться, как можно встроить потоковое видео в наш простой, но уже достаточно эффектный движок игры.

При воспроизведении потоковых данных в обработке информации может участвовать несколько фильтров. Один фильтр может декодировать данные и передавать другому фильтру, формировать данные в определенном формате для дальнейшего использования программой. В нашем случае будет создан фильтр. Он будет получать на входе декодированные данные и помещать их на поверхность текстуры, которую останется только использовать для натяжения на объект.

Создание фильтра достаточно непростая задача, но ее можно упростить, если использовать библиотеку базовых классов DirectShow, разработанную в Microsoft. Эту библиотеку вы можете найти в каталоге: DXSDK9\Samples \C++\DirectShow\ BaseClasses. Давайте не будем изобретать велосипед, а воспользуемся этой библиотекой, из которой нам понадобится класс `CBaseVideoRenderer`, на основе которого и будет строиться фильтр отображения видеоданных на текстуре.

Итак, для начала необходимо скомпилировать библиотеку, чтобы потом проще было ее использовать в собственных проектах и не приходилось каждый раз тащить за собой всю вереницу СРР-файлов. Для этого запустите проект baseclasses.dsw из каталога DXSDK9\Samples\C++\DirectShow \BaseClasses. Если вы запустите его из среды разработки, старше чем Visual Studio 6.0 (например, из Visual Studio 2003, как это делаю я), то перед вами появится приглашение конвертировать проект в новый формат (рис. 8.3). Если вы с таким окном еще не встречались, то советую нажать кнопку **Yes To All** (Да для всех).

Двигаемся дальше. Теперь для каждой необходимой вам конфигурации откомпилируйте проект. В проекте уже созданы конфигурации Debug и Release для стандартного и Unicode-проекта, и все их можно увидеть в раскрывающемся списке **Solution Configuration** (Конфигурация проекта) (рис. 8.4).

После компиляции проекта вы получите файл библиотеки strmbasd.lib или strmbase.lib, в зависимости от конфигурации. Скопируйте эти файлы в каталог с проектом, где вы собираетесь создавать проект с использованием фильтров потокового видео.

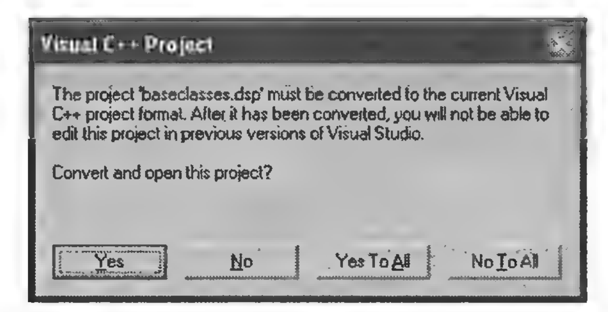


Рис. 8.3. Предупреждение о необходимости конвертировать проект в новый формат

Рис. 8.4. Раскрывающийся список **Solution Configuration** со списком конфигураций

8.2.2. Разработка фильтра

Теперь переходим непосредственно к разработке фильтра. Для этого необходимо создать "наследника" от `CBaseVideoRenderer`. Наследники этого класса используются для отображения видеоданных, что нам и нужно, нашего мы назовем `CTexFilter` и он будет располагаться в файлах TexFilter.cpp и

TexFilter.h. Чтобы вы лучше понимали структуру создаваемого нами класса, в листинге 8.1 показан заголовочный файл TexFilter.h.

Листинг 8.1. Содержимое файла TexFilter.h

```
#include <windows.h>
#include "d3d9.h"
#include "d3dx9.h"
#include "dshow.h"
#include "streams.h"

class CTexFilter : public CBaseVideoRenderer
{
 public:
  // Этот интерфейс понадобится нам для создания текстуры
  IDirect3DDevice9  *pDevice;
  // Текстура, на которой будем рисовать
  IDirect3DTexture9 *pTexture;

  D3DFORMAT Format; // Формат данных
  DWORD lVidWidth;  // Ширина поверхности/видео
  DWORD lVidHeight; // Высота поверхности/видео
  DWORD lVidPitch;  // Глубина цвета

  CTexFilter(IDirect3DDevice9 *pD3DDevice);
  ~CTexFilter();
  HRESULT CheckMediaType(const CMediaType *pmt);
  HRESULT SetMediaType(const CMediaType *pmt);
  HRESULT DoRenderSample(IMediaSample *pMediaSample);
  IDirect3DTexture9 *GetTexture(){return pTexture;};
};
```

Обратите внимание, что среди заголовочных файлов появились два новых для нас файла:

- dshow.h — это заголовочный файл DirectShow;
- streams.h — это заголовочный файл из состава базовых классов, которые мы собираемся использовать. В этом файле подключается все необходимое — все файлы библиотеки.

Чтобы проще было разобраться с переменными, которые объявлены в классе, я подробно закомментировал этот листинг. А вот с методами комментарии не помогут, потому что тут нужно слишком большие описания. Итак, помимо

конструктора и деструктора у нас здесь объявлены следующие методы (первые три перекрывают методы "предка"):

- CheckMediaType — этот метод вызывается, когда нужно подтвердить, что наш фильтр поддерживает или не поддерживает указанный тип данных;
- SetMediaType — в этом методе мы получаем установленный формат данных. Получив здесь размер картинки и глубину цвета, мы должны создать текстуру необходимого размера;
- DoRenderSample — это самый интересный метод, который вызывается, когда необходимо отобразить очередной кадр;
- GetTexture — метод просто возвращает указатель на текстуру, куда мы будем отображать данные. Так как метод прост и должен вернуть переменную pTexture, то реализовываем его прямо в заголовочном файле.

Конструктор достаточно элементарен — просто сохраняет указатель на интерфейс IDirect3DDevice9 и вызывает конструктор предка:

```
CTexFilter::CTexFilter(IDirect3DDevice9 *pD3DDevice)
     :CBaseVideoRenderer(__uuidof(CLSID_AnimatedTexture),
     NAME("ANIMATEDTEXTURE"), NULL, NULL)
{
       pDevice  = pD3DDevice;
}
```

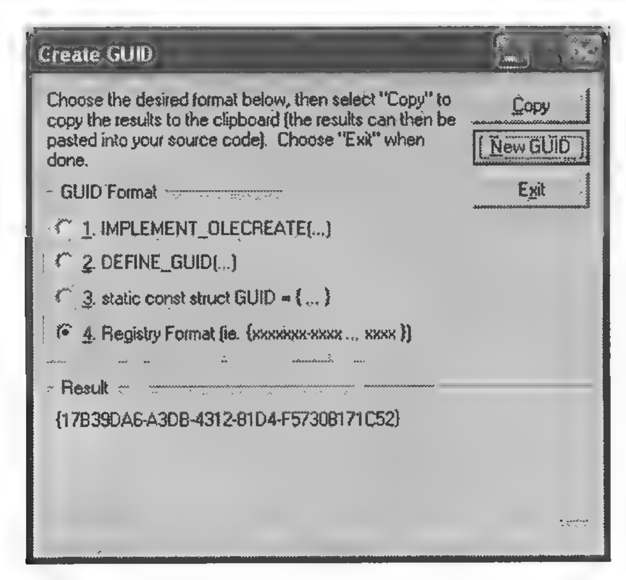


Рис. 8.5. Программа генерации уникальных идентификаторов

Конструктору предка необходим уникальный идентификатор CLSID_AnimatedTexture, который вы должны сгенерировать для вашего приложения. Это будет идентификатор класса вашего фильтра:

```
struct __declspec(uuid("{17B39DA6-A3DB-4312-81D4-F57308171C52}"))
   CLSID_AnimatedTexture;
```

для генерации уникального идентификатора GUID Gen, который можно найти в каталоге Tools (рис. 8.5).

Теперь давайте постепенно рассмотрим реализацию всех методов.

8.2.3. Подтверждение типов медиаданных

Как мы уже смогли разобраться, через метод CheckMediaType система может определить, поддерживаются ли фильтром загруженные данные. В нашем случае он выглядит следующим образом:

```
HRESULT CTexFilter::CheckMediaType(const CMediaType *pmt)
{
 // Проверяем указатель
 CheckPointer(pmt,E_POINTER);

 // Проверяем, чтобы была видеоинформация
 if(*pmt->FormatType() != FORMAT_VideoInfo)
  return E_INVALIDARG;

 // Тип должен быть Media, а подтип 24 бита
 if (IsEqualGUID(*pmt->Type(), MEDIATYPE_Video))
  if (IsEqualGUID(*pmt->Subtype(), MEDIASUBTYPE_RGB24))
   return S_OK;

 return E_FAIL;
}
```

В качестве параметра мы получаем экземпляр класса CMediaType, через который и определяется тип данных. В самом начале мы должны убедиться, что перед нами именно видеоданные. Для этого проверяем поле FormatType, и оно должно быть равно константе FORMAT_VideoInfo. О существующих форматах можно узнать в файле помощи. Например, для звуковых данных здесь будет константа FORMAT_WaveFormatEx.

После этого проверяем тип и подтип, т. е. результат вызова метода Type и Subtype. Тип должен быть равен MEDIATYPE_Video, а подтип MEDIASUBTYPE_RGB24. Мы будем работать только с 24-битными данными, чтобы упростить код формирования картинки на текстуре. Если этой проверки не делать, то

пользователь может подсунуть нам другой файл с другим форматом пиксела, который программа не сможет правильно обработать и завершиться аварийно.

Теперь посмотрим на код метода SetMediaType, который вызывается, когда тип медиаданных утвержден и здесь мы можем выполнить все необходимые подготовительные действия. Его код представлен в листинге 8.2.

Листинг 8.2. Код функции SetMediaType

```
HRESULT CTexFilter::SetMediaType(const CMediaType *pmt)
{
 D3DSURFACE_DESC ddsd;

 // Определяем размеры кадра
 VIDEOINFO *pviBmp = (VIDEOINFO *)pmt->Format();
 lVidWidth  = pviBmp->bmiHeader.biWidth;
 lVidHeight = abs(pviBmp->bmiHeader.biHeight);
 lVidPitch  = (lVidWidth * 3 + 3) & ~(3);

 // Создаем текстуру
 if (FAILED(pDevice->CreateTexture(lVidWidth, lVidHeight, 1, 0,
    D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, &pTexture, NULL)))
  return S_FALSE;

 // Получаем параметры созданной текстуры
 if (FAILED(pTexture->GetLevelDesc(0, &ddsd)))
  return S_FALSE;

 // Убедимся, что формат верный
 Format = ddsd.Format;
 if (ddsd.Format != D3DFMT_A8R8G8B8)
  return VFW_E_TYPE_NOT_ACCEPTED;

 return S_OK;
}
```

В качестве параметра мы снова получаем экземпляр класса CMediaType.

В самом начале метода определяем размеры видеоданных и глубины цвета. Эти данные нам понадобятся в этом методе для создания поверхности текстуры нужного размера, а во время формирования картинки для копирования данных на поверхность. Чтобы всегда держать необходимую информацию под рукой, сохраним размеры и глубину цвета в полях класса.

После этого создается текстура с помощью метода CreateTexture. Если все удачно, то получаем параметры текстуры и убеждаемся, что получился формат пиксела D3DFMT_A8R8G8B8. Если результат проверки отрицательный, то возвращаем VFW_E_TYPE_NOT_ACCEPTED, чтобы отклонить дальнейшее отображение видеопотока.

8.2.4. Получение кадра

В классе обработки видео нам осталось только перекрыть метод DoRenderSample, и можно считать, что необходимый минимум действий мы выполнили. Этот метод вызывается каждый раз, когда нужно получить очередной кадр с видеоданными. Видеоданные передаются в метод в виде интерфейса IMediaSample.

Однако, меньше разговоров и больше дела. В листинге 8.3 вы можете увидеть пример реализации получения видеоданных.

Листинг 8.3. Получение видеоданных

```
HRESULT CTexFilter::DoRenderSample(IMediaSample *pMediaSample)
{
 // Вспомогательные переменные для указателей буфера
 BYTE *pBmpBuffer, *pTxtBuffer;

 // Получить указатель на буфер с растровыми данными кадра
 pMediaSample->GetPointer(&pBmpBuffer);

 // Блокируем поверхность текстуры
 D3DLOCKED_RECT d3dlr;
 if(FAILED(pTexture->LockRect(0, &d3dlr, 0, 0)))
  return E_FAIL;

 pTxtBuffer =  static_cast<byte *>(d3dlr.pBits);
 LONG lTxtPitch = d3dlr.Pitch;

 BYTE  * pbS = NULL;
 DWORD * pdwS = NULL;
 DWORD * pdwD = NULL;
 DWORD col, row;
 DWORD dwordWidth = lVidWidth / 4;

 // Цикл перебора строк для копирования видеоданных
 for (row = 0; row< (UINT)lVidHeight; row++)
```

```
{
  pdwS = (DWORD*)pBmpBuffer;
  pdwD = (DWORD*)pTxtBuffer;

  // Цикл перебора колонок растровой картинки
  // для получения видеоданных
  for (col = 0; col < dwordWidth; col ++ )
  {
   pdwD[0] =  pdwS[0] | 0xFF000000;
   pdwD[1] = ((pdwS[1]<<8)  | 0xFF000000) | (pdwS[0]>>24);
   pdwD[2] = ((pdwS[2]<<16) | 0xFF000000) | (pdwS[1]>>16);
   pdwD[3] = 0xFF000000 | (pdwS[2]>>8);
   pdwD +=4;
   pdwS +=3;
  }

  pbS = (BYTE*) pdwS;
  // Цикл записи полученных данных на поверхность текстуры
  for (col = 0; col < (UINT)lVidWidth % 4; col++)
  {
   *pdwD = 0xFF000000 | (pbS[2] << 16) | (pbS[1] <<  8) | (pbS[0]);
    pdwD++;
    pbS += 3;
  }

   pBmpBuffer  += lVidPitch;
   pTxtBuffer += lTxtPitch;
  }

  // Разблокируем поверхность
  if (FAILED(pTexture->UnlockRect(0)))
   return E_FAIL;

  return S_OK;
}
```

Задача этой функции получить растровые данные кадра с помощью метода `GetPointer` и скопировать данные в нужное нам место. В нашем случае "нужным местом" является текстура. Если вы хотите вывести данные на экран, то можете просто копировать данные сразу в видеобуфер.

Хочу заметить, что в данном примере код копирования взят из примера, поставляемого в составе DirectX SDK — я такую чушь не писал ☺. Зачем же тогда я взял этот код, если он плохой? Чтобы разобрать ошибки, и вы потренировали свой мозг с целью написания более быстрого алгоритма.

Что страшного в этом коде? Конечно же, циклы, а их в этом примере аж три. Да, формат пиксела в источнике данных и приемнике разные, поэтому приходится копировать информацию попиксельно. Но три цикла это слишком. Просто цикл — это страшное дело, а здесь один основной и два вложенных. Первый вложенный запоминает растровые данные очередной строки в массиве, а второй цикл из этого массива копирует данные на поверхность текстуры. Неужели нельзя было сделать это одним циклом? В этом случае копирование сократилось бы почти в два раза.

Не удивительно, что MS-программы "глючат". Когда смотришь примеры или исходные коды, предоставляемые корпорацией, то поражаешься, как один кусок кода может быть написан гениально, а буквально рядом — небольшая бездарная процедура. Видимо уровень подготовки программистов в Microsoft слишком разный. Хочется надеяться, что в коде ОС таких проблемных мест в производительности минимум.

Итак, попробуйте избавиться от одного цикла и произвести копирование только двумя циклами, перебирая пикселы в ширину и высоту. Нужно только брать данные из источника и копировать их в приемник в одном цикле за один раз. Подобные задачи я не раз решал в книге [4] или [6].

8.2.5. Загрузка видео и использование фильтра

Теперь посмотрим, как можно использовать созданный нами код. Для этого в классе управления объектом `CDXGObject` нам понадобятся следующие переменные:

```
IGraphBuilder *pVideoGraph;
IMediaControl *pVideoControl;
```

Здесь у нас две переменные, которые имеют следующие типы интерфейсов:

- `IGraphBuilder` — с помощью этого интерфейса мы можем добавить свой фильтр в цепочку обработки потоковых данных;
- `IMediaControl` — интерфейс используется для контроля воспроизведением потоковых данных.

Существует еще несколько интерфейсов, которые могут применяться для управления потоковыми данными, но мы их не будем использовать. В большинстве случаев они вам не понадобятся.

Теперь посмотрим на код, который будет загружать видеоданные. Для этого в классе управления объектом добавим открытый (`public`) метод `LoadVideo`, который будет иметь следующий вид:

```
bool LoadVideo(char *filename);
```

Реализация метода показана в листинге 8.4.

Листинг 8.4. Функция загрузки видео

```
bool CDXGObject::LoadVideo(char *filename)
{
 // Локальные переменные
 WCHAR  wf[255];
 CTexFilter *pTexFilter = NULL;
 IBaseFilter *pFilter = NULL;
 IPin *pFilterPinIn = NULL;
 IBaseFilter *pSourceFilter = NULL;
 IPin *pSourcePinOut = NULL;

 // Создаем интерфейс IGraphBuilder
 CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
       IID_IGraphBuilder, (void **)&pVideoGraph);

 // Создаем экземпляр своего фильтра CTexFilter
 pTexFilter = new CTexFilter(pDevice);
 pFilter = pTexFilter;
 // Добавляем фильтр графопостроителю IGraphBuilder
 pVideoGraph->AddFilter(pFilter, L"DESTINATIONTEXTURE");

 // Указываем фильтр/файл источника
 mbstowcs(wf, filename, 255);
 pVideoGraph->AddSourceFilter(wf, L"SOURCEFILE", &pSourceFilter);

 // Связываем штейкеры фильтров
 pFilter->FindPin(L"In", &pFilterPinIn);
 pSourceFilter->FindPin(L"Output", &pSourcePinOut);
 pVideoGraph->Connect(pSourcePinOut, pFilterPinIn);

 // Получаем текстуру
 pTexture[0] = pTexFilter->GetTexture();

 // Получаем интерфейс контроля за потоком данных
 pVideoGraph->QueryInterface(IID_IMediaControl,  (void
**)&pVideoControl);

 // Если интерфейс контроля существует, то запускаем воспроизведение
 if (pVideoControl != NULL)
  pVideoControl->Run();
```

```
  // Освобождаем локальные переменные
  if (pFilterPinIn){pFilterPinIn->Release(); pFilterPinIn = NULL;};
  if (pSourceFilter){pSourceFilter->Release(); pSourceFilter = NULL;};
  if (pSourcePinOut){pSourcePinOut->Release(); pSourcePinOut = NULL;};

  return TRUE;
}
```

Давайте посмотрим, что у нас есть в этом листинге. В самом начале объявляем и обнуляем локальные переменные, которые будут использоваться в примере. Может быть, вы заметили, у меня привычка делать это в начале, которая связана слишком с частым просиживанием за Delphi.

После этого с помощью функции `CoCreateInstance` создаем экземпляр интерфейса `IGraphBuilder`. Это основной интерфейс, через который мы будем загружать файл и устанавливать фильтры.

После этого создаем экземпляр класса фильтра `CTexFilter`, который мы создавали, начиная с *разд. 8.2.2* до *разд. 8.2.4.* Теперь у нас есть фильтр, и мы можем добавлять фильтр источника и фильтр приемника. В качестве источника выступает файл и его необходимо добавлять с помощью метода `AddSourceFilter`. В общем виде этот метод выглядит следующим образом:

```
HRESULT AddSourceFilter(
  LPCWSTR lpwstrFileName,
  LPCWSTR lpwstrFilterName,
  IBaseFilter **ppFilter
);
```

Здесь имеется три параметра:

- `lpwstrFileName` — имя файла, данные которого необходимо использовать в качестве потоковых данных (в нашем случае видео);
- `lpwstrFilterName` — имя фильтра;
- `ppFilter` — фильтр, который будет создан на основе указанного файла.

Для добавления созданного нами выходного фильтра используется метод `AddFilter` интерфейса `IGraphBuilder`, который в общем виде выглядит следующим образом:

```
HRESULT AddFilter(
  IBaseFilter *pFilter,
  LPCWSTR pName
);
```

Здесь представлено два параметра:

- pFilter — фильтр, который необходимо установить. В нашем случае мы передаем указатель на экземпляр класса CTexFilter;
- pName — имя фильтра.

Теперь нужно связать входной фильтр (видеоданные, получаемые файлы) и выходной (наш фильтр, который должен получить данные). Это происходит с помощью следующего кода:

```
pFilter->FindPin(L"In", &pFilterPinIn);
pSourceFilter->FindPin(L"Output", &pSourcePinOut);
pVideoGraph->Connect(pSourcePinOut, pFilterPinIn);
```

Соединение двух фильтров происходит по принципу штекеров (Pin). Мы должны получить штекеры одного и другого фильтра и соединить их как бы кабелем. Для получения штекера используется метод FindPin, который в общем виде выглядит следующим образом:

```
HRESULT FindPin(
  LPCWSTR Id,
  IPin **ppPin
);
```

В качестве первого параметра необходимо указать имя, которое идентифицирует штекер. Существует два штекера:

- *входящий* (In, который получает данные);
- *выходящий* (Out, который возвращает результат обработки).

Второй параметр — это интерфейс IPin, через который мы получим результат работы.

Получив два штекера, мы соединяем их с помощью метода Connect интерфейса IGraphBuilder. Метод в общем виде выглядит так:

```
HRESULT Connect(
  IPin *ppinOut,
  IPin *ppinIn
);
```

Здесь представлена функция Connect, которая принимает два параметра. Оба они имеют тип IPin. Первый должен указывать на выходящий интерфейс, а второй на входящий. В результате метод соединит их.

Чтобы отображать объект, нам понадобится текстура. Для ее получения мы завели метод GetTexture(). И последнее, что нам потребуется — интерфейс IMediaControl, для управления потоковым видео. Его мы получаем с помощью метода QueryInterface следующим образом:

```
pVideoGraph->QueryInterface(IID_IMediaControl,
     (void **)&pVideoControl);
```

Если интерфейс управления получен, то запускаем воспроизведение с помощью метода Run:

```
if (pVideoControl != NULL)
 pVideoControl->Run();
```

Чтобы остановить воспроизведение, необходимо использовать метод Stop.

Теперь, чтобы при формировании сцены использовалась анимированная текстура с видеоданными в методе Render, можно использовать следующую логику:

```
if (pTexture[0])
{
 // Установить и использовать в шейдере текстуру pTexture[0],
 // которая содержит видеоданные
}
else if (pMeshTextures[0])
{
 // Использовать родную текстуру для сетки
}
```

Вот и все. Пример готов к использованию. Осталось только дать несколько замечаний по компиляции, и вы будете готовы увидеть результат.

8.2.6. Компиляция

Наш проект уже готов, и теперь мы можем приступать к компиляции и запуску приложения. В результате у нас должно получиться что-то похожее на рис. 8.6. Да, рисунок не может передать видео, которое воспроизводится на стене, и текстура получилась не очень хорошего качества, за счет MPEG-сжатия.

Перед компиляцией убедитесь, что у вас есть все необходимые библиотеки в параметре Additional Dependences свойств проекта. У меня этот параметр состоит уже из следующего набора библиотек: d3d9.lib, d3dx9.lib, d3dxof.lib dxguid.lib, dinput8.lib, strmiids.lib, strmbasd.lib, winmm.lib. Если у вас проект не компилируется, то, значит, вы что-то забыли добавить.

Пример получился достаточно полезным и, возможно, вы найдете ему применение в своих играх или графических проектах. Для того чтобы отобразить картинку на весь экран, например, чтобы показать игроку ролик между уровнями, достаточно только создать прямоугольник на весь экран и натянуть на него текстуру, отображающую видео.

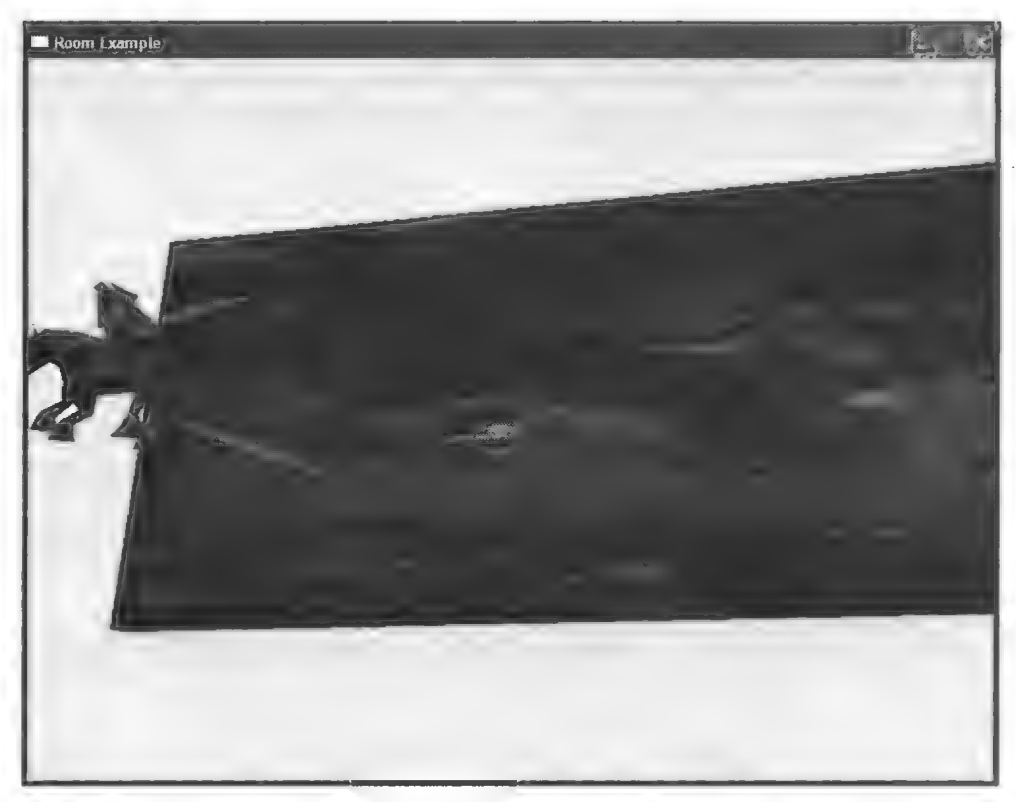


Рис. 8.6. Результат работы программы

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter8\Video.

8.3. Мелочи жизни

В игре очень часто можно встретить мелкие объекты разнообразного типа, которые в большом количестве путешествуют по экрану или даже игровому миру. К таким объектам можно отнести капли воды, которые могут капать с крыши, снаряды и другие объекты.

Как поступить с такими объектами, которых должно быть много? Создавать каждый снаряд из вершин? Если пулю сделать в виде куба, то понадобится 8 вершин, а в массовой перестрелке количество пуль может быть до сотни, т. е. лишние 800 вершин или 600 граней, на каждую из которых нужно натянуть текстуру, чтобы пуля выглядела более реалистичной. Это вам надо? Видеопроцессору точно это не нужно.

Можно попытаться обмануть пользователя, как было описано в *разд. 8.1*, но это экономия только половины вершин и останется еще 400, а реалистичность падает достаточно серьезно. Есть какие-то предложения, как серьезно сократить количество вершин? Нет, сокращение количества мелких объектов не принимается к обсуждению, потому что они необходимы и придаются сцене реалистичности, динамичности и т. д. Есть более полезные решения?

Мелкие объекты на столько мелкие, что их можно делать из вершин. Да, мы должны просто научиться правильно визуализировать вершины. Почему правильно? Большинство книг по DirectX рассказывает нам о том, что вершины можно выводить на экран и что им можно задавать цвет, но при этом авторы очень часто забывают сказать нам, что на вершины можно натягивать текстуру. Как это? Ведь вершина — это всего лишь точка? Да, это точка, которая имеет размер, и если сделать точку размером в сантиметр (имеется в виду размер в ширину, высоту и глубину, ведь увеличенная точка представляет собой куб), то получится достаточно большой куб, на который можно натянуть текстуру. А если еще и включить прозрачность и правильно ее использовать, то выйдет отличный результат.

Посмотрите на рис. 8.7, где показаны четыре точки с натянутой текстурой. Красиво? Нет, вы не можете ощутить всей красоты на черно-белой картинке. Чтобы ощутить шок, советую запустить пример из каталога Samples\Points на прилагаемом компакт-диске. Этот пример я рассматривал в книгах [4] и [6]. Шок в том, что это действительно всего лишь 4 точки.

Давайте кратко "пробежимся" по примеру, чтобы понять, как все устроено.

Для начала увеличиваем размер точки на 1.02, по сравнению со значением по умолчанию:

```
DWORD ps;
pD3DDevice->GetRenderState(D3DRS_POINTSIZE, &ps);
pD3DDevice->SetRenderState(D3DRS_POINTSIZE, ps*1.02);
```

После этого включаем альфа-смешивание:

```
pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

Ну и самое главное — устанавливаем свойства отображения точек:

```
pD3DDevice->SetRenderState(D3DRS_POINTSPRITEENABLE, TRUE);
pD3DDevice->SetRenderState(D3DRS_POINTSCALEENABLE, TRUE);
```

В первой строке мы разрешаем натягивать на точку текстуру. Для этого параметр `D3DRS_POINTSPRITEENABLE` устанавливаем в значение `TRUE`. Во второй

строке включаем параметр D3DRS_POINTSCALEENABLE, который разрешает масштабировать точки.

Теперь если установить текстуру, то она будет отображена на вершине:

```
pD3DDevice->SetTexture(0, pTexture);
```



Рис. 8.7. Точка с натянутой на нее текстурой

Вот и все. Вот таким образом мы с помощью только одной вершины можем добавить в игровую сцену достаточно красивый и интересный объект.

8.4. Не все золото, что блестит

Мы уже ни один раз использовали шейдеры для повышения производительности примеров или для создания в движке таких эффектов, как освещение или тени. Вершинные и пиксельные шейдеры позволяют реально повысить производительность работы графической программы.

В этом разделе мы поговорим о шейдерах с точки зрения создания игровых объектов и придания им красивого вида. Посмотрите на рис. 8.8, где показан скриншот (screen shot) комнаты, в котором все объекты, освещение и тени

отображаются через шейдеры. Конечно, черно-белая полиграфия не может передать всей красоты, поэтому этот же рисунок можно найти на компакт-диске в каталоге Samples\room.bmp, а в каталоге Samples\ATI находится сам пример, который написан программистами компании ATI.



Рис. 8.8. Пример использования шейдеров от компании ATI

В этом разделе мы будем разбирать различные шейдеры и надеюсь, что они пригодятся вам, даже если вы не собираетесь писать собственные игры. Итак, пора погружаться в мир красоты и наслаждаться им, ведь именно красота обязана спасти мир.

Давайте рассмотрим один пример, в котором с помощью шейдера создается очень красивая поверхность объекта. Создать шелковую поверхность достаточно сложно без использования шейдера. Посмотрите на шелк при ярком свете, и вы увидите одновременно и отражения, и преломления, и другие эффекты игры со светом. В листинге 8.5 показан шейдер, который создает эффект шелковой поверхности на объекте.

Листинг 8.5. Шейдер создания шелковой поверхности

```
// Матрицы, участвующие в расчете
float4x4 view_proj_matrix;
float4x4 view_matrix;

// Переменные
float4 color =  {1, 0.3f, 1, 1};
float4 gloss=  {0.7f, 0.5f, 0.4f, 1};
float4 lightDir =  {-0.4f, 0.4f, -0.8f, 0};
sampler Noise;

// Структура входных данных вершинного шейдера
struct VS_INPUT
{
  float4 Pos: POSITION;
  float3 normal: NORMAL;
  float3 tangent: TANGENT;
};

// Структура выходных данных вершинного шейдера
struct VS_OUTPUT {
   float4 Pos:      POSITION;
   float3 normal:   TEXCOORD0;
   float3 tangent:  TEXCOORD1;
   float3 viewVec:  TEXCOORD3;
   float3 pos:      TEXCOORD4;
};

// Вершинный шейдер
VS_OUTPUT vertsh(VS_INPUT In)
{
  VS_OUTPUT Out;

  Out.Pos = mul(In.Pos, view_proj_matrix);

  Out.normal   =  mul(In.normal, view_matrix);
  Out.tangent  =  mul(In.tangent, view_matrix);

  Out.viewVec  = normalize(-mul(In.Pos, view_matrix));
  Out.pos = In.Pos.xyz;

  return Out;
}
```

```
// Входные данные пиксельного шейдера
struct PS_INPUT
{
  float3 normal: TEXCOORD0;
  float3 tangent: TEXCOORD1;
  float3 viewVec: TEXCOORD3;
  float3 pos: TEXCOORD4;
};

// Пиксельный шейдер
float4 pixsh(PS_INPUT In) : COLOR
{
   float angle = (tex3D(Noise, In.pos) - 0.5);
   float cosA, sinA;
   sincos(angle, sinA, cosA);

   float3 tang = sinA * In.tangent + cosA * In.normal;

   float diffuse = saturate(dot(lightDir, In.normal));
   float cs = -dot(In.viewVec, tang);
   float sn = sqrt(1 - cs * cs);
   float cl = dot(lightDir, tang);
   float sl = sqrt(1 - cl * cl);
   float specular = pow(saturate(cs * cl + sn * sl), 32);

   return diffuse * color + gloss * specular;
}

// Техника отображения
technique PixelLight
{
    pass P0
    {
        VertexShader = compile vs_2_0 vertsh();
        PixelShader  = compile ps_2_0 pixsh();
    }
}
```

На рис. 8.9 можно увидеть результат отображения сферы с использованием этого шейдера. Цветная копия этого рисунка находится на компакт-диске в файле Screen1.bmp каталога \Samples\Shelk.

Вершинный шейдер достаточно прост. Здесь всего лишь корректируем позицию, нормаль и касательную с учетом положения текущей вершины. Самое интересное кроется в пиксельном шейдере. Здесь берется точка с текстуры

Noise и ее цвет математически корректируется, чтобы получить этот эффект шелка. Текстура Noise — это не та текстура, которая должна отображаться на поверхности объекта, а специально подготовленное изображение. Чтобы получить максимально реалистичный шелк, я рекомендую использовать рисунок шума или что-то похожее на море или небо.



Рис. 8.9. Результат работы программы

С помощью шейдеров можно украшать не только поверхности сеток, но и точки, которые мы рассматривали в *разд. 8.3*. Я стараюсь, и вам рекомендую, использовать шейдеры везде, где это только возможно, потому что это работает быстрее, удобнее и эффективнее. Чтобы изменить внешность объекта или алгоритм отображения, не нужно изменять код программы. Достаточно изменить только файл с шейдером и никакой компиляции не требуется.

Примечание

Исходный код проекта из этой главы находится на компакт-диске в каталоге Chapter8\Look.

8.5. Искусственный интеллект

Искусственный интеллект — это отдельная и достаточно интересная тема. Нет, мы не будем сейчас ничего реализовывать, потому что именно искусственного интеллекта я никогда не разрабатывал, поэтому ничего утверждать не могу. У меня просто нет достаточного опыта, чтобы учить других. Но с точки зрения оптимизации могу предложить некоторые решения.

В игре просто обязаны быть противники, которые должны как-то двигаться по комнате и реагировать на вас. Чтобы игра была реалистичной, каждый противник должен обладать *искусственным интеллектом* и лучше всего, если этот интеллект будет разный. Для этого можно запрограммировать несколько сценариев поведения монстров и раздавать их объектам в зависимости от типа.

Искусственный интеллект должен выполнять следующие действия:

1. Объекту нужно перемещаться по виртуальному миру, причем не просто от стенки к стенке, а с каким-то смыслом. Он может где-то останавливаться, задумываться и приседать. Но если в вашей игре все монстры стоят, каждый за своим углом, и начинают двигаться только в тот момент, когда вы подходите к стене, то такая игра становится слишком прямолинейной и некрасивой.
2. Объекты должны как-то реагировать на ваши действия, причем не тупо бежать с монтировкой на вашу базуку, а если необходимо то прятаться, отступать или идти в обход.

Это минимум выполняемых противником действий, но этот минимум слишком накладный для процессора. Если в игре будет 1000 монстров, то перед каждым отображением сцены необходимо перебрать всех монстров и обработать их интеллект, т. е. если нужно, то передвинуть персонаж по комнате или выполнить определенное действие. Один цикл из 1000 шагов достаточно накладный, а тут еще на каждом шаге нужно выполнять какие-то операции перемещения объекта, а это потребует и проверки столкновения объекта со стенами, в результате искусственный интеллект отнимет все возможные ресурсы процессора.

Если сделать все объекты неподвижными, пока они не попадут в поле зрения игрока, то проблема решается сама собой. Единственное, что нужно сделать перед расчетом искусственного интеллекта, — это проверить, видим ли объект. Конечно же, эту проверку не обязательно делать с точностью до пиксела, можно просто проверить, находится ли объект в той же комнате, что и вы. А т. к. в одной комнате более 10 противников не располагают, то обработка интеллекта будет не такой уж и долгой. Но тут нужно учитывать, что если

открыта дверь, то в смежной комнате монстры тоже должны что-то делать, например:

- если произошел выстрел, то монстры из соседней комнаты должны побежать на шум, даже если все двери в комнату закрыты. Вы же не будете объяснять пользователю, что все комнаты в вашем виртуальном мире с великолепной звукоизоляцией;
- если игрок забежит и выбежит из комнаты, то монстры не должны замораживаться, иначе это будет слишком наглядно, особенно если монстр успел вас заметить и начал преследовать;
- если вы просто перемещаетесь по комнате на достаточно большом расстоянии от противника, который стоит к вам спиной, то он не должен на вас реагировать. У него же простые уши, а не локаторы, поэтому при расчете искусственного интеллекта учитывайте, в какую сторону повернута голова противника.

Этот список можно продолжать, но мы остановимся. Надеюсь, что вы уже поняли, что необходимо быть предельно внимательным и аккуратным.

Чтобы запрограммировать все это, понадобится достаточно много времени. Недаром в фирмах по разработке игр искусственным интеллектом может заниматься отдельный программист, который оттачивает движок до совершенства. Хотя нет, совершенства не бывает и не может быть, но к нему необходимо стремиться.

Если же вы решили разработать игру самостоятельно, то придется позаботиться обо всех нюансах, но лучше что-то упростить, иначе игра никогда не будет закончена. Если вы решили сделать искусственный интеллект максимально приближенным к реальности, то могу посоветовать поискать по этой теме литературу. Конкретные издания не посоветую, потому что не читал подобных работ, но книги по данной теме видел в Интернете.

Итак, давайте немного поговорим в теории о том, как можно оптимизировать обработку искусственного интеллекта. Во-первых, структура, описывающая каждый объект, должна состоять примерно из следующих полей:

- матрицы, определяющей позицию объекта;
- вектора, определяющего направление взгляда;
- количества жизни;
- X и Y координаты позиции на игровом поле;
- время последней обработки логики.

Это примерный список полей, который в реальной жизни может быть больше или меньше. Обратите внимание, что есть два параметра, определяющие позицию объекта в мире — матрица и координаты X и Y. Зачем нужно второе,

если с помощью матрицы можно без проблем позиционировать объект в виртуальном пространстве? Секрет кроется в том, что вы должны разбить все на квадраты, т. е. наложить на ваш мир сетку. Сетка может представлять собой разбиение на комнаты, но лучше разделить мир так, как нашу планету делят меридианы.

Чтобы было проще, можно сделать сетку 10×10 или 100×100. При этом, чтобы узнать в каком именно квадрате находится сейчас персонаж, достаточно разделить позицию на 10:

```
int Xblock = WorldPosMatrix._41 % 10;
int Yblock = WorldPosMatrix._42 % 10;
```

В данном случае `WorldPosMatrix` — это матрица, определяющая позицию объекта. В первой строчке кода мы определяем блок по оси X, а вторая строка определяет блок по оси Y.

В определенный момент ваш персонаж может находиться в одном из квадратов мира. Например, на рис. 8.10 показан пример, где черный квадрат определяет позицию объекта, а серыми квадратами помечены смежные области.

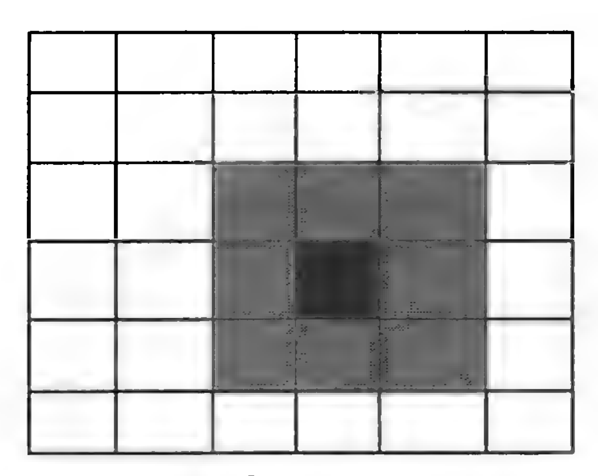


Рис. 8.10. Разбиение мира на квадраты

Теперь логика обработки искусственного интеллекта должна быть следующей:

- если объект находится в белом квадрате, то он статичен и обработки его логики не будет;
- если объект в сером квадрате, то он может быть не виден, но его логику лучше обработать и это можно делать раз в секунду (реже не стоит), а не перед каждым формированием сцены. При этом нужно учитывать время, которое прошло с момента последней обработки. Если во время определе-

ния логики оказалось, что данный монстр находится в области видимости (вполне реальная ситуация, когда два объекта из разных квадратов видны друг другу), то делать обработку логики при каждом формировании сцены;

- если объект в белом квадрате, то его обработку необходимо производить перед каждым отображением сцены.

Зачем нужно обрабатывать смежные квадраты? Если персонаж находится почти на границе квадрата, то он может увидеть противников, которые расположены в соседнем квадрате. И если расстояние между противником и персонажем меньше, чем область видимости, то противник должен реагировать. Ведь два объекта игры могут находиться в разных квадратах, но при этом смотреть друг другу в глаза, если оба находятся на границе.

Вот тут есть еще один нюанс — в вашей игре будет какой-то массив, который будет хранить все объекты игры. Перед каждым отображением сцены "пробегать" по всему этому массиву с проверкой и определять, в каком квадрате находится противник — достаточно накладно. Я рекомендую "пробегать" по этому массиву только в тех случаях, когда ваш персонаж пересекает границу квадрата, т. е. переходит в другой квадрат. В этот момент необходимо просмотреть информацию обо всех противниках, и если кто-то находится в нужном или смежном квадрате, то скопировать его данные в буфер. Таким образом, в буфере будут находиться немного противников, и только те, которых надо просматривать во время обработки искусственного интеллекта.

8.6. Оптимизация графики

Сейчас мы немного "пробежимся" по принципам оптимизации 3D-графики. Графика — это сфера, в которой оптимизации никогда не бывает много, т. е. над совершенствованием производительности программы можно работать бесконечно. На протяжении всей книги я достаточно часто ссылался на производительность и стремился показать вам проблемные участки и места, где могут возникнуть проблемы и где стоит уделить внимание оптимизации. В данном случае мы затронем только 3D, но не забывайте, что еще есть оптимизация циклов, долго выполняющихся функций и т. д. Эти принципы справедливы и для домашних утилит или приложений для работы с базами данных. Однако такую тему мы опустим, тем более что ее я уже описывал в некоторых разделах, ранее изданных книг — [1], [4] и [6].

При создании фигуры с помощью Direct3D используйте минимально необходимые размеры. Например, при создании буфера индексов `IDirect3DIndexBuffer9` каждый элемент массива может быть 16- или 32-битным. Если количество индексов не превышает 65535, то следует использовать 16-битный массив. В данном случае переход на 32 бита будет не оправданным и понесет лишние расходы памяти, которой никогда не бывает много.

Конечно же, на скорость работы влияют и размеры экрана. Чем больше размеры, тем больше ресурсов необходимо для расчетов сцены. Получается, что и здесь мы должны находить разумную достаточность, т. е. получить необходимое качество изображения и при этом достаточную производительность, чтобы зритель (пользователь) не видел сильных задержек во время формирования сцены.

Оптимизация графики — это достаточно интересный процесс, потому что здесь помимо общих принципов оптимизации программ существует еще множество приемов. Например, чтобы оптимизировать размер картинки, можно ее сжать классическим архиватором. Размер файла уменьшится, но коэффициент сжатия зависит от самой картинки, например, фотографии с большим количеством цветов сжимаются очень плохо.

Человеческий глаз не способен различить два оттенка цвета, если в них отличается на единицу только одна составляющая. Получается, что если объединить близлежащие точки с небольшим отличием в оттенке и заменить их одним цветом, то файл сожмется больше, а пользователь ничего и не заметит. Такое сжатие называется *компрессией с потерей качества*. Тут нужно добиваться компромисса между качеством изображения и качеством сжатия и это достигается путем выбора максимальной разницы между оттенками, которые можно объединить. Ведь если будут объединены два цвета, разница которых незаметна глазу, то качество пострадает не сильно. Но если отличие заметно, то качество теряется.

Где это используется? Конечно же в файлах изображений, которые может загружать программа. Чем меньше файл, тем быстрее его можно загрузить в память с жесткого диска, который является самым слабым звеном компьютера. После этого процессор восстановит изображение, и если алгоритм сжатия не сложный, то это может быть выполнено быстрее, чем загружать с диска не сжатые данные.

Количество используемых цветов влияет и на требуемые ресурсы. Чем больше цветов мы используем, тем больше нужно памяти для хранения изображений и поверхности и больше нужно процессорного времени, чтобы копировать всю эту память между поверхностями. Предположим, например, что у нас есть картинка размером в 100×100 пикселов. Если используется глубина цвета в 1 байт, то для хранения изображения понадобится 10000 байт памяти. Поскольку такая картинка сможет содержать максимум 256 цветов, а это очень мало, и поэтому желательно использовать минимум 2 байта, когда количество цветов будет равно 65535. Этого уже достаточно для хранения более качественного изображения, но потребуется 20000 байт памяти, что в два раза больше, а значит, и копирование данных будет требовать от процессора в два раза больше ресурсов.

Если же выбрать глубину цвета в 24 бита, то тут уже понадобится в 3 раза больше ресурсов, хотя количество цветов будет исчисляться 16777216. Вот и думай после этого, что выбрать — качество или скорость. Наша задача выбрать оптимальный вариант, который позволит получить лучшее соотношение скорости и качества.

Размер изображений также играет немаловажную роль. Ведь если мы выбрали разрешение экрана в 800×600 пикселов при 16-битном цвете, то для хранения поверхности потребуется $800 \times 600 \times 2 = 960000$ байт памяти. Это почти 1 Мбайт! А если изображение будет 1024×768, то тут уже объем необходимой поверхности памяти составит 1572864 байт. Снова увеличение ресурсов в два раза, а значит, и падение производительности во время копирования содержимого поверхностей.

Оптимизация графики на первоначальном этапе — это борьба выбора между качеством и скоростью. В настоящее время стандартом стали ЖК-мониторы с диагональю 15 дюймов, и чтобы картинка на них выглядела приемлемо, необходимо использовать разрешение 800×600, а лучше 1024×768.

Но неплохо было бы сделать возможность выбора необходимого разрешения. Например, у меня широкоформатный ноутбук и идеалом для него является разрешение 1280×800, а изображение при разрешении 800×600 и 1024×768 выглядит растянуто, что очень неудобно. Поэтому в данном случае можно предложить пользователю самому выбирать необходимое разрешение в зависимости от имеющихся ресурсов.

В качестве глубины цвета мы чаще всего будем использовать 16 бит, потому что оно позволяет добиться приемлемого качества при минимуме затрат. Можно предложить пользователю выбирать и глубину цвета, но тогда придется писать слишком универсальный код, который понизит производительность. Чтобы не терять в скорости, лучше все же привязаться к определенной глубине экрана.

В Direct3D заметить потерю качества цвета будет еще сложнее, если сцена будет находиться в движении. Все мы видели фильмы на DVD, где качество картинки не соответствует 100%, но вы заметили это несоответствие? Я думаю, что нет. Потерю качества можно увидеть, только если нажать паузу в момент быстро сменяющейся смены сцены, например, во время быстрого движения на автомобиле. Лучше будет, если действия будут происходить в темноте (ночью). В этом случае происходит максимальное сжатие и можно даже увидеть разводы на черном фоне ночи.

Подведем итог — для демо-роликов лучше использовать глубину цвета 16 бит и давать пользователю самому выбирать разрешение экрана, в зависимости от которого зритель сможет контролировать качество и скорость изо-

бражения. В играх пользователь может остановиться и рассмотреть стену или небо, и если там будут откровенные ляпы, то это плохо скажется на его мнение об игре.

В Direct3D мы формируем сцену с помощью вершин и треугольников. Чем больше их, тем больше времени необходимо движку и видеокарте на формирование сцены. И снова приходится выбирать между количеством и качеством. Посмотрите на рис. 8.11, где показана сфера, созданная из 32-х сегментов. Сфера получается гладкой, но количество необходимых треугольников слишком велико.

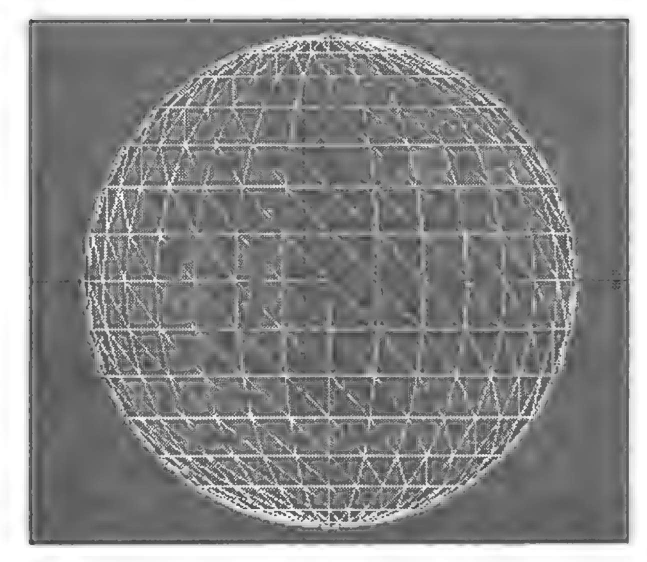


Рис. 8.11. Сфера из 32-х сегментов

Можно сократить количество сегментов до 16, т. е. уменьшить их в два раза. В этом случае сокращается количество ресурсов, необходимых для отработки фигуры, но сфера получается угловатой (рис. 8.12). Я думаю, что такое положение дел устроит далеко не многих.

Если же нам нужны сглаженные поверхности, то приходится выбирать между гладкостью и скоростью. В большинстве игр стараются использовать прямоугольные поверхности, ведь для создания четырехугольной плоскости достаточно всего лишь двух треугольников.

Получается, что скорость создания сцены зависит не только от количества объектов на экране — объектов интерьера, освещений, существования теней. На сцену влияют и количество вершин, из которых состоит объект. Здесь необходимо подходить к решению задачи с деликатной осторожностью.

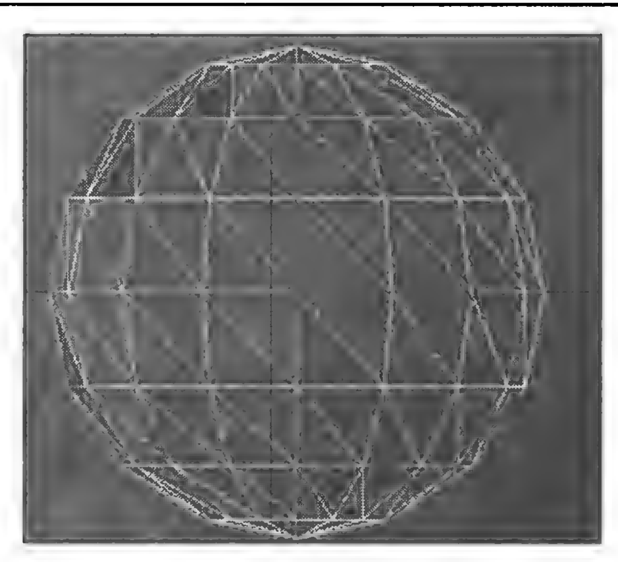


Рис. 8.12. Сферв из 16-ти сегментов

С другой стороны, если создать сферу из 100 сегментов, то это будет явно избыточно, особенно если эта сфера в сцене будет находиться очень далеко и выглядеть мелкой. Пользователь просто не сможет увидеть мелких деталей, которые вы захотите передать. Если у вас есть 3D-редактор, позволяющий контролировать количество сегментов (например, 3D Studio Max), то попробуйте сейчас создать сферу размером во весь экран. Для нормального отображения и получения гладкой поверхности необходимо не менее 30 сегментов. Но если отдалить сферу как можно дальше (чтобы она была диаметром не более десятикопеечной монеты), то будет достаточно и 12-ти сегментов.

Получается, что при формировании сцены мы можем использовать разное количество сегментов для объектов с различной удаленностью и значительно сэкономить жизнь процессору и видеокарте, а значит, увеличить скорость создания сцены. Мы постараемся в данной книге использовать этот эффект для повышения производительности.

Посмотрите на рис. 8.13, где показаны сферы различного размера. Самая большая сфера создана из 32-х сегментов, а остальные из 12-ти. Обратите внимание, что самая маленькая сфера выглядит вполне гладкой, хотя на тех сферах, что побольше, хорошо видны угловатости. Использовать большое количество сегментов на маленьких объектах — излишняя растрата ресурсов. Пример, который формирует эту сцену, можно найти на компакт-диске в каталоге \Samples\Sphere.

То же самое касается и текстур. Чем ближе объект, тем лучше видна каждая деталь на текстуре. На самой большой сфере рис. 3.8 можно увидеть, что текстура есть, но на самой дальней сфере вы ничего не увидите. Она на столько

мала, что кажется окрашенной в один цвет. Тогда зачем использовать текстуру? Не лучше ли просто окрасить сферу в нужный цвет и пользователь ничего не заметит, особенно если сфера будет в движении.

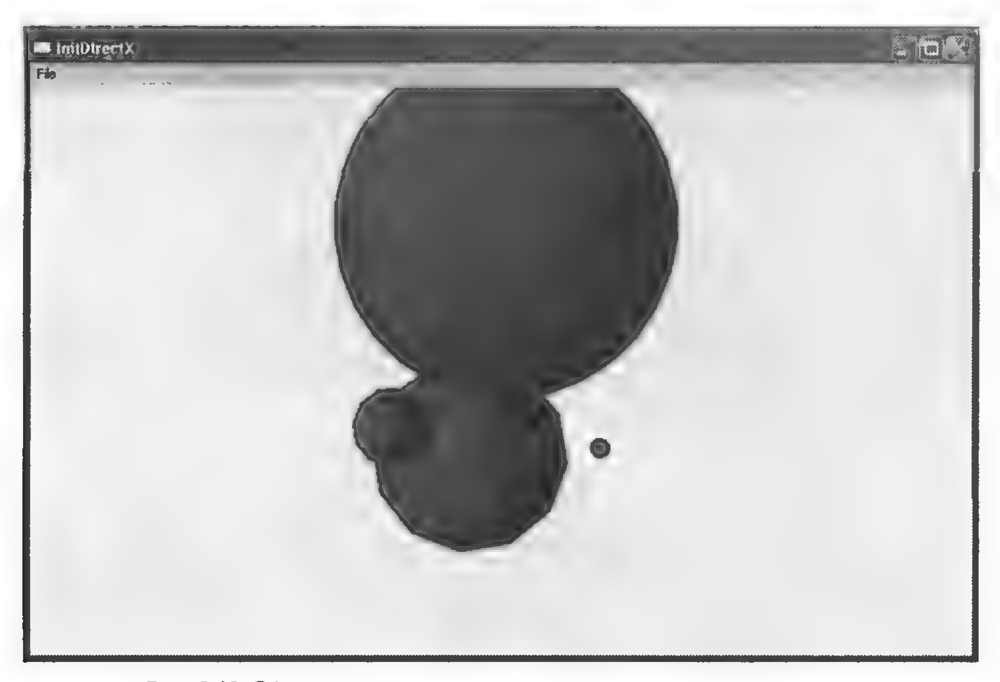


Рис. 8.13. Сферы разного размера и из разного количества сегментов

Чтобы повысить скорость работы с текстурами, можно действовать следующим образом:

- для самых дальних объектов можно вообще не использовать текстуры, а просто окрашивать объект цветом, который будет максимально соответствовать цвету текстуры;
- по мере приближения объекта, можно натянуть на него текстуру небольшого размера, например, 16×16. Таким образом, мы заранее масштабируем битовый файл, облегчая жизнь видеокарте;
- для больших объектов, находящихся поблизости, можно использовать более качественные текстуры.

Если масштабировать текстуры заранее, то качество изображения будет лучше, чем если это сделает DirectX, потому что профессиональный художник сможет сделать необходимые сглаживания, чтобы уменьшенное изображение выглядело приемлемо. Конечно, все возможные варианты текстур подготовить невозможно, но хотя бы несколько вариантов подготовить нужно.

Меньшие по размеру текстуры, которые будут использоваться для дальних объектов, будут обрабатываться быстрее за счет меньшего размера и меньшего коэффициента программного масштабирования. Заранее масштабированные текстуры — это один из немногих трюков, когда мы повышаем и скорость, и качество сцены.

На большинство внутренних процессов Direct3D мы не можем оказать влияние, но есть кое-что, на что повлиять можно. Например, с помощью функции OptimizeInplace можно задать контроль отображения точек и поверхностей сеток (mesh) для оптимизации производительности. Эта функция в общем виде выглядит следующим образом:

```
HRESULT OptimizeInplace(
    DWORD Flags,
    CONST DWORD *pAdjacencyIn,
    DWORD *pAdjacencyOut,
    DWORD *pFaceRemap,
    LPD3DXBUFFER *ppVertexRemap
);
```

Рассмотрим параметры этой функции:

- Flags — параметры, определяющие действия, которые необходимо выполнить для оптимизации. Здесь можно указать следующие значения:
  - D3DXMESHOPT_COMPACT — пересортировать поверхности сетки, чтобы убрать не используемые вершины и поверхности;
  - D3DXMESHOPT_ATTRSORT — пересортировать поверхности для уменьшения атрибутов;
  - D3DXMESHOPT_VERTEXCACHE — пересортировать поверхности для повышения коэффициента эффективности использования кэша вершин;
  - D3DXMESHOPT_STRIPREORDER — пересортировать поверхности для увеличения длины близлежащих треугольников;
  - D3DXMESHOPT_IGNOREVERTS — оптимизировать только поверхности, не затрагивая вершины;
  - D3DXMESHOPT_DONOTSPLIT — пока сортируются атрибуты, не разделять вершины, которые разделяются между группами атрибутов;
  - D3DXMESHOPT_DEVICEINDEPENDENT — оптимизировать размер кэша вершин;
- pAdjacencyIn — указатель на оптимизированный массив смежных граней;
- pAdjacencyOut — указатель на оптимизированный массив смежных граней;
- pFaceRemap — указатель на выходной буфер, куда будут помещены новые индексы для каждой поверхности;

- ppVertexRemap — указатель на интерфейс ID3DXBuffer, который будет содержать новые индексы для каждой вершины.

Эту функцию можно вызывать сразу после загрузки каждой сетки. Например, в нашем случае ее можно добавить в конец функции LoadMesh, которую мы рассматривали в *разд. 1.5* и использовали для загрузки сеток в наших проектах. Даже простая сортировка уже может повысить производительность:

```
(ID3DXMesh *)(*ppMesh)->OptimizeInplace(D3DXMESHOPT_ATTRSORT,
   NULL, NULL, NULL, NULL);
```

Во время оптимизации графики необходимо знать меру. Как мы говорили в начале раздела, оптимизация может быть с потерей качества и без. Сначала необходимо попытаться повысить производительность программы без потерь в качестве. Когда вы жертвуете качеством, то не забывайте, что игра может потерять все прелести. Можно пойти следующим способом — сделать в настройках игры пункт, который позволит включать высокое качество или нет. Например, можно разрешить пользователю включать освещение, тени и другие технологии, использование которых может отрицательно повлиять на скорость работы программы.

В этом разделе мы рассмотрели только основы оптимизации, и только то, что касается графики. Но не забываем, есть еще много принципов оптимизации, которые действуют в приложениях любого типа. Например:

- оптимизация циклов и сокращения операций, выполняющихся много раз;
- оптимизация процедур. На вызов каждой процедуры уходит достаточно много времени (сохранение параметров в стеке и переход по указанному адресу), поэтому если есть возможность, то от лишних процедур лучше отказываться;
- в качестве параметров процедур, имеющих большой размер, передавать адреса, а не значения;
- можно использовать оптимизацию компилятора, ведь все современные компиляторы могут оптимизировать код с целью создания минимального размера кода или максимальной скорости выполнения.

Этот список можно продолжать, но я уже не раз говорил о методах оптимизации. Здесь же я хочу остановиться на еще одной очень важной особенности игр, которую часто упускают из виду большинство программистов — минимальные требования. Дело в том, что не секрет, что львиная доля игр создается на C++ и чаще всего для компиляции используется компилятор от Microsoft. Да, это признанный стандарт, и кто, как не производитель ОС лучше оптимизирует программу для работы в Windows. Но так ли нужна эта оп-

тимизация в отношении Windows? С полной уверенностью говорю — не нужна!!! Почему? Об этом буквально в следующем абзаце.

Меня поражают паразиты, которые разрабатывают программы (в том числе и игры) с минимальными требованиями к процессору Pentium 4 и при этом компилируют с помощью Visual C++ 6.0 без патчей и обновлений. Что тут поразительного? А то, что Visual Studio 6.0 без патчей не знает о существовании Pentium 4, он даже команды MMX по умолчанию не использует. Почему не использовать того, что есть в минимальных требованиях? Одни только команды MMX могут повысить производительность, а если включить еще SIMD, то можно выиграть в производительности до 20%, не внося ни капли изменений в код.

Теперь можно сказать, почему не нужна оптимизация под ОС Windows? Игры минимально используют возможности ОС, а больше применяют DirectX и процессор, а кто, как не Intel лучше знает архитектуру своих процессоров? По некоторым тестам, одна только смена компилятора может повысить производительность программы на 10—15%, особенно в программах, интенсивно рассчитывающих графику. Да, такие показатели можно увидеть не очень часто, а может быть и замедление работы, но 10% вполне реальная цифра, и это абсолютно без каких-либо вмешательств в код. Главное — используйте процессор по максимуму и задействуйте все возможности.

Остановитесь и загляните сейчас на Web-сайт Intel **http://www.intel.com /cd/software/product/asmo-na/eng/compilers/cwin/index.htm** и почитайте информацию о компиляторе (рис. 8.14).

После того, как игра готова, запускаем ее и ищем самое слабое место. Я всегда искал его "на глаз" (в принципе, в игровом движке это не так уж и сложно определить слабое место), но недавно познакомился с программой Intel VTune Performance Analyzer. Главное окно этой программы можно увидеть на рис. 8.15. Эта утилита предназначена для анализа скорости выполнения программы. Настраиваем VTune, запускаем программу и через какое-то время смотрим, какие замеры сделал VTune. Основное, что нас интересует — какая функция работает дольше всех и именно ее оптимизацией нужно заниматься в дальнейшем.

Но не торопитесь — одного только времени выполнения мало. Дольше всех может выполняться функция инициализации игрового уровня, где происходит загрузка текстур и т. д., но ее оптимизацией заниматься не обязательно. Необходимо учитывать прежде всего те функции, которые участвуют непосредственно в рендеринге.

Помимо этого, необходимо учитывать и количество вызовов. Например, одна функция может выполняться достаточно быстро, но вызываться очень часто.

Проанализируйте, если вызов происходит из одного или двух мест, то может быть лучше сделать ее inline, т. е. избавиться от нее вовсе и сэкономить лишний переход, возврат, передачу параметров и т. д. В сумме это будет экономия в районе трех и более команд (зависит от количества параметров), но если умножить это значение на количество вызовов функции, то экономия становится существенной. Например, если за секунду произошло 1000 вызовов функции, то если избавиться от нее, то экономия составит 3000 команд в секунду, а это уже не так уж и плохо.

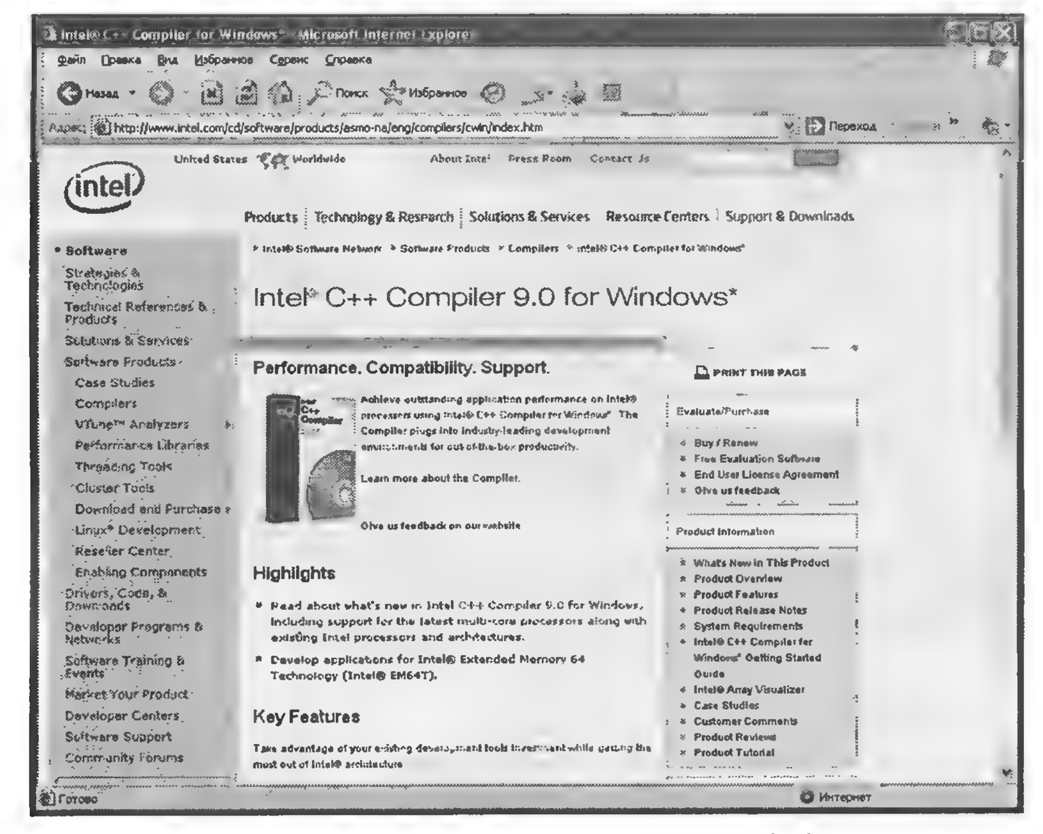


Рис. 8.14. WEB-сайт с информацией о компиляторе Intel

Кстати, в играх вообще не должно быть маленьких функций, выполняющих от одного до трех простых действий. Именно они чаще всего вызываются по сотни раз и съедают драгоценные такты.

Корпорация Intel заботится о разработчиках и разработчиках игр в частности. На сайте intel.com можно найти множество полезной информации о различных методах оптимизации и эффективных способах использования возмож-

ностей современных процессоров. Сейчас можно найти множество документов по использованию Hyper-Threading в играх, а ведь двухъядерные процессоры действительно могут повысить производительность и на порядок. Если для игры два ядра являются минимумом, то почему не воспользоваться ими для своих нужд?

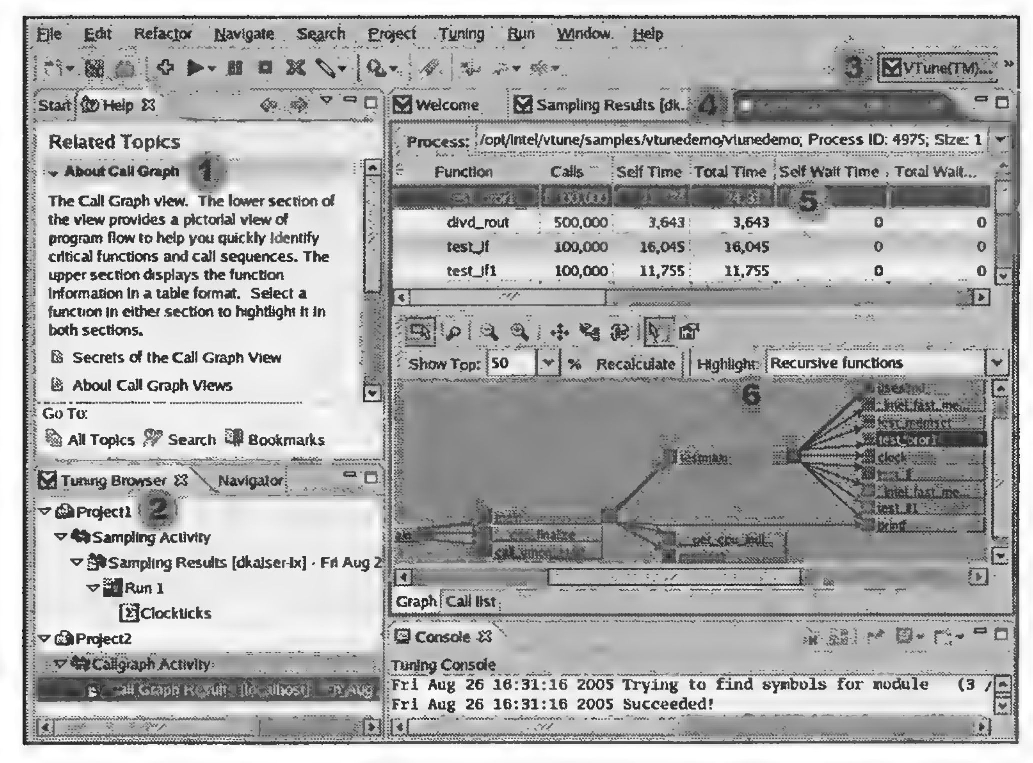


Рис. 8.15. Главное окно программы VTune

На сайте есть информация и на русском языке. Да, ее намного меньше, чем на английском, но основные и последние документы переводятся (рис. 8.16). Советую заглянуть на следующую страничку:

http://www.intel.com/cd/ids/developer/emea/rus/dc/games/index.htm

Компании ATI и NVidia являются законодателями моды на рынке домашних видеоускорителей и так же, как и Intel, заботятся о разработчиках. На их сайтах можно найти тонны информации о разработке графических эффектов, игр и использовании максимальных возможностей последних видеочипов.

Чем сильнее вы нагрузите видеочип, тем быстрее будет работать программа. Почему? Чаще всего видеоускорители выполняют задачи быстрее, чем центральный процессор, потому что они специально заточены для графических

вычислений, матриц, векторов и т. д. Некоторые задачи могут выполняться параллельно с вычислениями центральным процессором, и это так же повысит производительность.

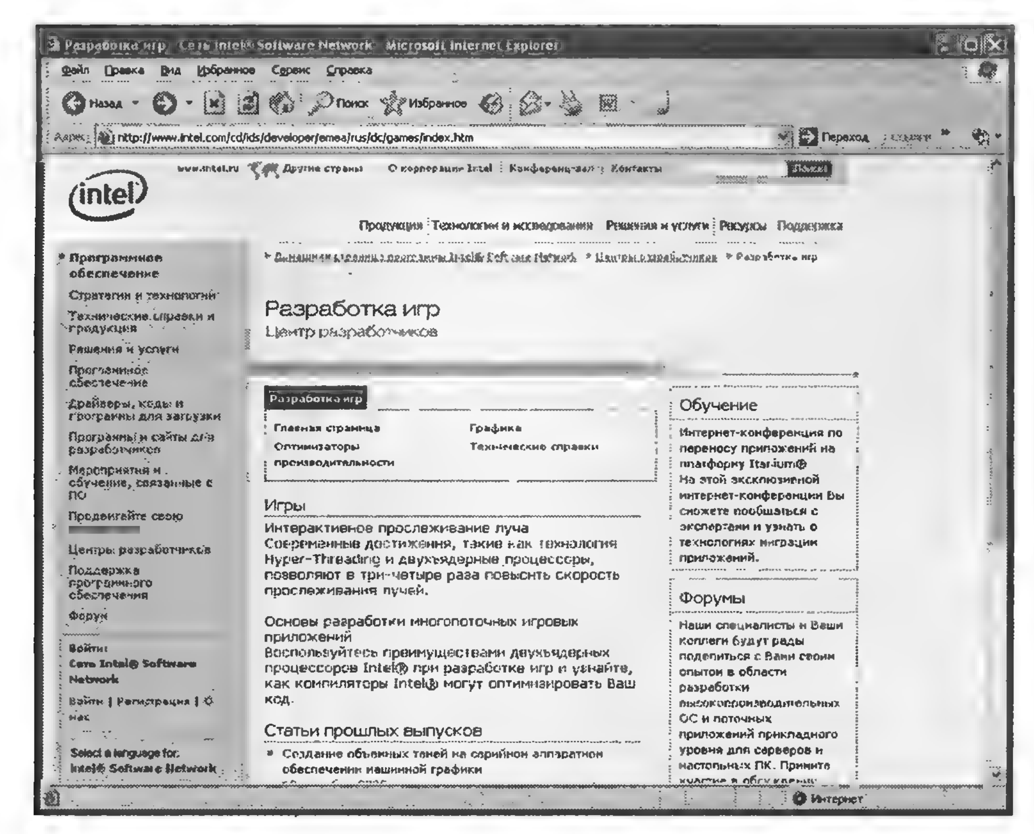


Рис. 8.16. Раздел для разработчиков игр на сайте Intel

Всю свежую информацию для разработчиков игр от компании NVidia можно найти на сайте **http://developer.nvidia.com**.

Если вы решили использовать какую-то информацию с сайтов **www.ati.com** или **www.nvidia.com**, то будьте осторожны. Некоторые функции могут работать на видеочипе одного производителя, но вызвать серьезные проблемы на чипе другого производителя. В этом случае можно потерять большую часть потенциальных пользователей, а если проект коммерческий, то и покупателей. Чтобы решить эту проблему, можно создать две сборки исполняемых файлов, одна для чипа ATI и другая для чипа NVidia. Это сделать не так уж и сложно. Нужно создать две функции для разных SDK и подключать их в зависимости от выполняемой в данный момент сборки. Да, усложняется отладка, но прямой вызов функций драйвера видеокарты лучше.

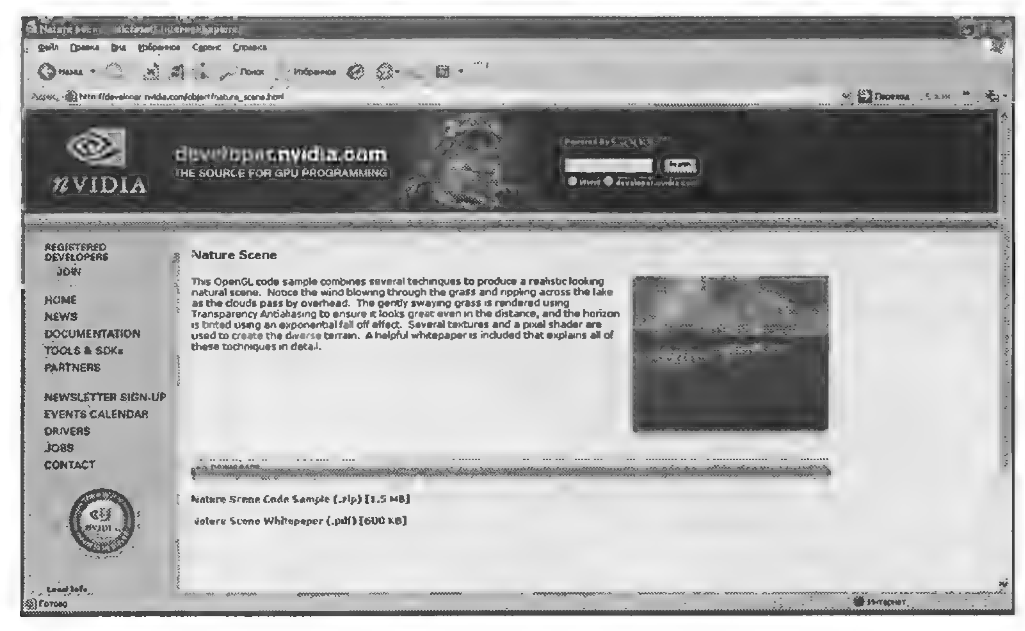


Рис. 8.17. Раздел для разработчиков игр на сайте NVidia

Заключение

На протяжении всей книги мы постепенно рассматривали процесс создания игрового движка и на его примере анализировали возможные решения проблем, с которыми может столкнуться разработчик игры. Рассмотреть все возможные проблемы в такой маленькой книге нереально, да и я не имею такого опыта создания игр, чтобы встретиться с этими проблемами. В данном случае мы рассматривали простой движок, в стиле 3D-шутеров, но есть еще ролевые игры, стратегии, аркады и другие жанры, в которых есть свои особенности.

Рассмотренный нами движок достаточно прост и далек от идеала, но я его создавал не для реальной игры, а для иллюстрации возможностей DirectX, для создания игр и для иллюстрации возможностей библиотеки.

Программирование игр — очень интересное занятие, потому что здесь всегда есть поле деятельности для мозгов. В 3D-графике одну задачу можно решить несколькими способами, но опытные разработчики игр всегда советуют решать проблему наиболее быстрым и эффективным методом. Не нужно на все 100% следовать физике. Достаточно сделать ваш мир похожим на натуральный, а небольшую (именно небольшую, а не размером с трамвайную остановку ☺) погрешность никто не заметит. Наша задача решить проблему максимально эффективным методом и с минимальной погрешностью. Главное, чтобы ноги не проваливались под пол, и не было откровенных ошибок физики. Виртуальный мир должен хоть немного быть похожим на натуральный, а как вы этот мир создаете, никого не волнует.

Уже давно известно, что успех игры на 90% зависит от идеи игры, т. е. насколько она интересна, а не от графики. Графическое и музыкальное сопровождение — это всего лишь дополнение к игре. Вот если вы собираетесь конкурировать на рынке 3D-шутеров и сюжет будет основываться на классическом — беги, стреляй, то тогда придется делать упор на графику и музыку. Только в этом случае кто-то купит игру и побегает по вашему виртуальному миру, любуясь его красотами и эффектами. Но даже в этом случае бег будет

недолгим, потому что даже суперкрасивый мир быстро надоест, поэтому не забывайте подкреплять графический движок вашей игры хоть каким-то смыслом, а лучше уникальным.

Хотелось бы дать еще множество советов, но я не могу сказать, что я гуру в игровом мире. Я поделился своими знаниями с точки зрения программирования, но вы, наверное, уже давно поняли, что игра это не только код. Я мог бы попытаться дать вам советы и с точки зрения музыки или художественного оформления, но это уже будут советы не профессионала, а игрока. И тут один совет все же напрашивается сам собой — прислушивайтесь к игрокам, а чтобы узнать их мнения, что они хотят, читайте обзоры в игровых журналах и на сайтах, посвященных играм. Будьте ближе к народу, и он к вам потянется. Изучайте своих конкурентов, и стремитесь выделиться на фоне остальных, иначе вы можете остаться серой мышкой в жестоком мире информационных технологий.

Ваши комментарии и пожелания я жду по e-mail издательства "БХВ-Петербург" **mail@bhv.ru** — работники издательства перешлют мне, а лучше на форуме сайта **www.vr-online.ru**. Единственная просьба — старайтесь задавать вопросы коротко, но четко описывайте проблему, потому что я получаю каждый день очень много писем, и только их чтение отнимает много времени. Если решение вашей проблемы можно описать в нескольких предложениях, то я отвечу, если нет, то постараюсь дать направление, в котором необходимо двигаться, чтобы самостоятельно решить проблему. Если я долго не отвечаю, не отчаивайтесь, видимо мой ящик просто переполнен, или меня нет дома. Я стараюсь отвечать всем.

Если у вас есть какие-то решения по улучшению описанного движка, и вы готовы поделиться своей работой с миром, то я с удовольствием приму их по почте и выложу на своем сайте, благо места на нем достаточно.

ПРИЛОЖЕНИЕ

Описание компакт-диска

Папки	Описание
\Chapter1	Исходные коды *главы 1*. Пример загрузки сетки Mesh и базовое приложение
\Chapter2	Исходные коды *главы 2*. Создание базовых возможностей движка
\Chapter3	Исходные коды *главы 3*. Примеры скелетной анимации
\Chapter4	Исходные коды *главы 4*. Пример работы с устройствами ввода
\Chapter5	Исходные коды *главы 5*. Примеры определения столкновений
\Chapter6	Исходные коды *главы 6*. Вершинная анимация
\Chapter7	Исходные коды *главы 7*. Примеры работы со звуком
\Chapter8	Исходные коды *главы 8*. Пример работы с видео и использование шейдеров для получения красивого эффекта
\Soft	Демонстрационные программы от CyD Software Labs
\Common	Общие файлы, используемые в проектах
\Sample	Дополнительные примеры
\Demo	Игра kkrieger
\Doc	Дополнительная документация
\Media	Медиафайлы, используемые в примерах книги
\URL	Файлы URL-ссылок для быстрого доступа к сайтам, описанным в книге

Список литературы

1. Фленов М. Программирование на C++ глазами хакера. — СПб.: БХВ-Петербург, 2004. — 350 с.
2. Фленов М. Программирование в Delphi глазами хакера. — СПб.: БХВ-Петербург, 2003. — 370 с.
3. Фленов М. Компьютер глазами хакера. — СПб.: БХВ-Петербург, 2005. — 336 с.
4. Фленов М. DirectX и C++. Искусство программирования. — СПб.: БХВ-Петербург, 2006. — 384 с.
5. Кулагин Б. и Морозов Д. 3ds max 6 и character studio 4. Анимация персонажей. — СПб.: БХВ-Петербург, 2004. — 224 с.
6. Фленов М. DirectX и Delphi. Искусство программирования. — СПб.: БХВ-Петербург, 2006. — 384 с.

Предметный указатель

К

М

О

П

С





Ц

Ш

Я

ИСКУССТВО ПРОГРАММИРОВАНИЯ ИГР НА C++

«... Если ваша мечта – создание 3D-шутера, то она может стать реальностью... Описываемый движок очень прост и позволяет создавать игры и другого жанра...»
Михаил Фленов

Когда-то мы все играли. Целлулоидные голыши, железные конструкторы, монструозные заводные игрушки. Мы играли в эти социалистические изобретения и временами создавали что-то свое – выжигали по готовым деревянным чудищам, выпиливали грузовички на уроках труда, раскрашивали их дома. И даже теперь, когда производители заваливают нас игрушками – как реальными, так и виртуальными, мы не перестаем творить. Мы пишем для собственного удовольствия, чтобы самосовершенствоваться и заставить мозг работать, чтобы узнать какие-то технологии, порадовать ближних или заработать денег. Да-да, в эпоху мобильных устройств и новой жизни старых игр (которые создавались разработчиками-одиночками и маленькими командами) это стало возможно. Я уверен, что данная книга поможет тебе в этом.
Александр Лозовский,
выпускающий редактор журнала «ХакерСпец», редактор журнала «Хакер»

Фленов Михаил,
профессиональный программист. Работал в журнале «Хакер», в котором несколько лет вел рубрики «Hack-FAQ» и «Кодинг для программистов», печатался в журналах «Игромания» и «Chip-Россия». Автор бестселлеров «Библия Delphi», «Программирование в Delphi глазами хакера», «Программирование на C++ глазами хакера», «Компьютер глазами хакера», «DirectX и Delphi. Искусство программирования», «DirectX и C++. Искусство программирования» и др. Некоторые книги переведены на иностранные языки и популярны в США, Канаде и других странах.

3D-игры притягивают игроков атмосферой виртуальной реальности, а программистов интересными алгоритмами, положенными в основу работы программ. Разработка игры связана с серьезной тренировкой ума, и в этой книге вы найдете некоторые интересные решения для создания собственного игрового движка, который будет использовать все современные технологии: вершинные и пиксельные шейдеры, скелетную и вершинную анимацию, а также компоненты DirectMusic, DirectSound и DirectInput, входящие в библиотеку DirectX. Поскольку этот движок реализует только базовые возможности игры, его нельзя назвать полноценным, однако он может стать хорошей пищей для размышления, а код, описываемый в книге, можно легко адаптировать и превратить в полноценную игру.

Компакт-диск содержит листинги из книги и дополнительную информацию по DirectX.

ISBN 5-94157-832-6





БХВ-ПЕТЕРБУРГ
194354,
ул. Есенина, 5Б
E-mail: mail@bhv.ru
internet: www.bhv.ru
тел./факс: (812) 591-6243